DESIGNERS AND THEIR TOOLS:

COMPUTER SUPPORT FOR DOMAIN CONSTRUCTION

By

TAMARA R. SUMNER

B.S., University of California, Santa Cruz, 1982

B.A., University of California, Santa Cruz, 1985

M.S., University of Colorado at Boulder, 1992

A thesis submitted to the Faculty of the Graduate School of the University of Colorado in partial fulfillment of the requirement for the degree of Doctor of Philosophy

Department of Computer Science

1995

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1995		**00-00-1995 to 00-00-1995**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Designers and Their Tools: Computer Support for Domain Construction	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Department of Computer Science,University of Colorado,Boulder,CO,80309**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**216**	

This thesis for the Doctor of Philosophy degree by

Tamara R. Sumner

has been approved for the

Department of Computer Science

by

Gerhard Fischer

Mike Eisenberg

Date _____

Dissertation Committee:

Gerhard Fischer,	Computer Science
Clayton Lewis,	Computer Science
Mike Eisenberg,	Computer Science
Mike King,	U S WEST Advanced Technologies
Peter Polson,	Psychology
Mark Gross,	Environmental Design

Designers and their Tools: Computer Support for Domain Construction

Tamara R. Sumner (Ph.D., Computer Science)

Thesis directed by Professor Gerhard Fischer

ABSTRACT

In today's high-technology workforce, many designers work in dynamic and innovative domains such as user interface and software design. This thesis considers the problem of providing these designers with computational design support tools. It differs from other efforts in that the emphasis is on understanding and supporting the evolutionary patterns of change inherent in dynamic domains. The motivating problem investigated is flexibility, specifically: *How can systems provide domain-specific support for short-term design activities yet still be flexible enough to accommodate long-term evolutionary patterns of change in the domain?* Three empirical studies show how design communities gradually construct their domain by defining important domain objects, creating and evolving representations for viewing these objects, and establishing relationships between objects and representations. The observed design process is termed *domain construction*. Analyses of these studies are used to characterize the observed domain construction processes of *use, elaboration,* and *modification*. Design environments based on two computational models are analyzed in terms of these domain construction processes. The two computational models occupy seemingly different ends of the spectrum from support to flexibility. The Toolbelt model, where practitioners assemble and evolve collections of generic software applications, seemingly offers more flexibility. The Domain-Oriented Design Environment model, where customized domain-oriented tools are created for a specific community, seemingly provides better support but is less flexible. These intuitions are investigated to better understand which aspects of a computational model (i.e., underlying data models, domain models, architectures, and integration services) help or hinder an environment's overall flexibility. The results are used to develop recommendations for the next generation of design support environments.

ACKNOWLEDGEMENTS

I'd like to thank the following individuals for their help and support:

Michael Wright for good food, good company, and moral support.

Gerhard Fischer for providing the opportunity and resources to pursue this research.

The designers at U S WEST Advanced Technologies: Susan Davies, Mike King, Josh Staller, Jason Webb, Lynda Baines, and Bruce Keahy (without them, very little of this work would have been possible).

My "unofficial" dissertation committee: John Rieman, Gerry Stahl, Markus Stolze, and Alex Repenning.

Several people who contributed much to VDDE: Benedikte Harstad, Nathalie Bonnardel, and Alex Repenning.

Chris DiGiano for his patience in explaining multimedia design.

The UU Group: Jonathan Ostwald, Stefanie Lindstaedt, Kurt Schneider, Markus Stolze, and Kumiyo Nakakoji

Tom "Toolbelt" Yoksas for posing with his favorite toys.

The members of L3D for a fun and productive working atmosphere.

This research was supported in part by U S WEST Advanced Technologies and ARPA under grant No. N66001-94-C-6038. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author (s) and do not necessarily reflect the views of the agencies named in this section.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis considers the problem of providing small, independent design teams working in innovative and dynamic domains with computational design support tools. This research differs from other efforts in that the emphasis is on *understanding and supporting the evolutionary patterns of change inherent in dynamic domains*. This dynamic aspect places new challenging burdens on computational design support tools. Specifically, in innovative and dynamic domains, design tools must:

(1) Meet the traditional goal of providing direct support for individual design activities, and

(2) Meet the new goal of being flexible enough to support design communities to continually evolve their design practices to support changing domain needs.

These two goals pose a challenging system design problem because they are diametrically opposed. Providing direct support for individual design activities usually requires enriching a system with an understanding of the task or the domain; i.e., building a domain model into the system. However, building a rich domain model into a system often limits the system's flexibility in that it is no longer capable of dealing with significant, or even small, changes in the domain. The purpose of this research is to investigate computational architectures, extension mechanisms, and interfaces that enable practitioners in design communities to create and progressively modify design support tools that meet these two paradoxical goals.

MOTIVATING PROBLEM

In today's workforce, many design professionals work in emerging high-technology domains such as computer network design, user interface design, or multimedia title design. These designers are usually members of small, independent, cross-functional teams, with diverse educational and organizational backgrounds, that come together only for the duration of a specific project or related set of projects. Projections on the workforce of the next century indicate that design teams such as these will become increasingly prevalent in the years ahead [85, 90]. Thus, there are very pragmatic reasons why it is important that the tools used by these designers effectively support their design activities and enable them to create better designs more efficiently.

In their daily design activities, these designers face *traditional problems* due to the inherently open-ended [92] and ill-structured [109] nature of design. For any design problem, there is an infinite number of possible solutions and no single correct answer. Additionally, designers need to look into the future and envision how their artifact will be used before it is built in order to evaluate the benefits and limitations of their approach [51]. Finally, designers must integrate knowledge from multiple domains in order to arrive at a solution [15, 49]. Typically, this knowledge does not reside in the head of any single person but instead, is distributed across other team members involved in the design process [101]. A large part of the designer's job is communicating various aspects of the design under construction to other design stakeholders so that they may contribute their respective expertise and ideas.

There are numerous research efforts focused on creating design support environments that address these traditional design problems [10, 22, 25, 50, 59, 99, 111]. A commonalty across these research efforts is that the systems advocated provide specialized task-specific or domain-oriented design representations. Domain-oriented systems embody a model of the entities to be manipulated and the tasks to be performed and use the model to provide active support to the user. An example is

a kitchen design environment enabling users to construct floorplans from objects such as cabinets and stoves, which can then be analyzed by the system for compliance with design guidelines [29]. Proponents of these approaches claim these specialized design representations provide better cognitive support for the design activity because users interact with familiar entities and do not need to build up domain entities from other lower-level operations [28, 80, 99]. Additionally, the rich domain models embedded in these systems enable them to directly support design tasks by providing simulation facilities and active knowledge-based assistance [10, 29, 30, 33, 76, 78, 108, 111].

However, designers also face *new problems* in that innovative domains are characterized by rapid and continual change as underlying technologies and design practices evolve. For instance, in the mid-1980's most user interfaces were based on character-oriented command lines. By 1990, graphical user interfaces based on windows and direct manipulation techniques were prevalent. Now, in the mid 1990's, alternate modalities are starting to gain prominence such as voice dialog interfaces and interfaces based on 3-D virtual reality metaphors. In effect, the domain of user interface design has undergone at least three major revolutions in technology and design practices in the last 10 years. Such periodic *revolutionary upheavals* are important components of evolutionary change in innovative and dynamic domains.

In these domains, evolutionary change also involves continual *incremental elaborations and refinements*. Sometimes, incremental changes are in response to a minor innovation in technology such as the introduction of a new pointing device in the realm of user interfaces. Many times, these changes are brought about as designers continually strive to improve their design practices. Many of these designers are engaged in product design activities and in the new, highly-competitive global economy there is continual pressure to make better products and to shorten product lifecycles [85]. Towards this end, these designers are organizationally empowered, and indeed many times required, to make significant decisions concerning their own

design process. In effect, part of their job is devising better ways to do their work and they are rewarded for continually improving their process.

It is not clear that domain-oriented software is flexible enough to address these new design problems. Specifically, the specialized design representations and embedded domain model may need to be continually modified to reflect changing conditions just to keep pace with the rate of evolutionary change in dynamic domains. In fact, recent empirical studies indicate that professionals need generic tools that are flexible enough to support a wide range of practices and representations. Generic tools are applications like word processors, graphics packages and databases seem to be more flexible because they do not focus on a particular task or domain. Instead, these applications support the creation and manipulation of a particular type of representation, such as documents or drawings, by providing a wide range of formatting features and flexible editing commands. Nardi and Johnson found that professional slidemakers preferred collections of generic graphic tools to slide making-specific software because of the generic tools' greater power and flexibility [81]. Sumner also observed how the flexibility of generic tools enabled designers to evolve design representations to better support changing work practices [116]. However, this same study also showed that practitioners were trying to use generic applications in very domain-specific ways; the tools' generic nature had several negative side effects such as introducing cognitive and manual burdens on constructing and maintaining designs and hindering iterative design.

In summary, designers in innovative and dynamic domains need computational support tools because they are an important part of the future economy and it would behoove us to provide them with better tools to do their job. However, creating software tools to support these designers is challenging because what constitutes the domain is continually evolving in response to changes in technology, the surrounding business milieu, and design practices. On the one hand, these designers could benefit from the support provided by domain-

oriented tools; on the other hand , they seem to need the flexibility inherent in generic tools. Thus, the motivating problem investigated in this thesis is how to strike a balance between supporting *use* and supporting *change*. Particularly, the question considered is:

> How can systems provide direct (i.e., domain-oriented) support for individual, daily design activities *yet still be flexible enough* to accommodate evolutionary patterns of change (i.e., periodic revolutionary upheavals and continual incremental elaborations) in the domain and design practices that occur over time?

APPROACH

The general approach taken to investigate this question combines empirical studies, development of theoretical frameworks, and system building and assessment. Figure 1.1 shows the empirical and theoretical foundations, the core contributions, and the results of this research.

First, empirical studies of designers are used to characterize and analyze the patterns of change in a specific innovative and dynamic domain – user interface design. The results of these studies, combined with existing theories of design, provide the motivation for the domain construction theoretical framework. This framework builds upon existing theories of design and organizational communication to characterize how cognitive, communicative, and social aspects of design interact to drive evolutionary changes within a design domain. A detailed analysis of domain construction processes suggests the functionality that design environments *and their underlying computational models* must provide in order to effectively support designers working in innovative and dynamic domains.

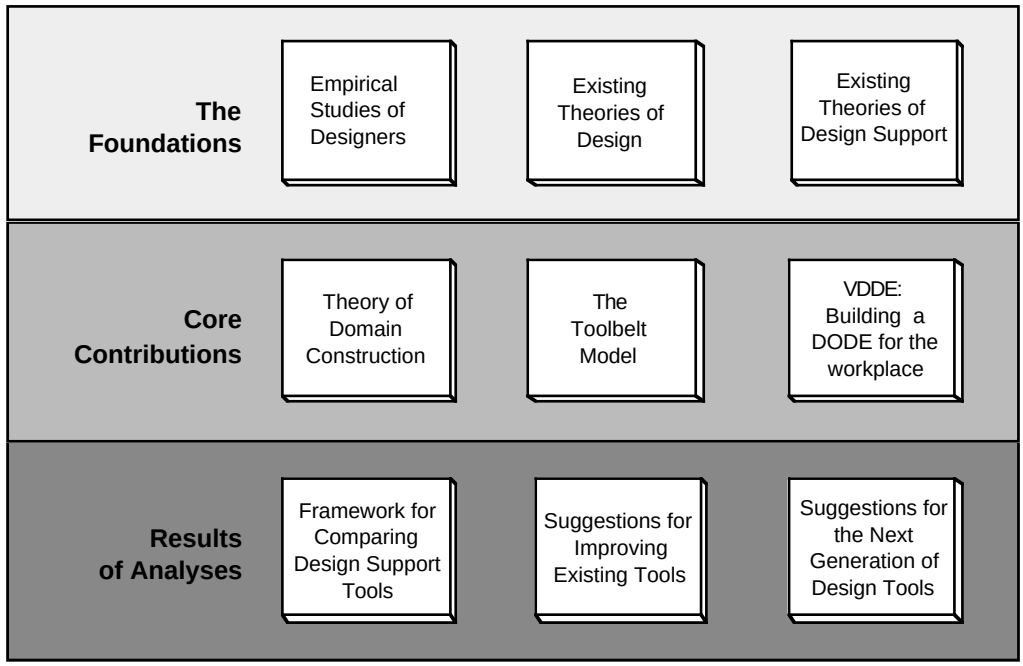

Figure 1.1: The structure of this research.

This framework is then used to analyze and compare two computational models at seemingly different ends on the support / flexibility spectrum. The Toolbelt model, where practitioners assemble and evolve collections of generic software applications, seemingly offers more flexibility. The Domain-Oriented Design Environment (DODE) model, where customized domain-oriented tools are created for a specific community, seemingly provides better support but is less flexible. The purpose of this analysis is to investigate these intuitions in order to better understand which particular aspects of a model help or hinder its use and overall flexibility.

Two specific design support systems instantiating each of these models were developed to support a particular group of user interface designers during the course of this research. One system, based on the Toolbelt computational model, was created over time by the user interface designers themselves and characterized and studied as part of this research [116]. The other system , the Voice Dialog Design Environment (VDDE) [99, 117], is based on the domain-oriented design environment computational model [25]. This system was

collaboratively constructed by myself and the same user interface designers concurrently with the development of the toolbelt model.

These two different design support systems – Toolbelts and DODEs – are individually analyzed and then compared according to the domain construction framework. This analysis particularly looks at the flexibility of each model; examining how well various aspects of the underlying computational model (e.g., architecture, object model, and tailoring mechanisms) support the different domain construction processes. The results of these analyses are used to derive recommendations for improving each individual model and to motivate recommendations for the next generation of design support systems.

WHO SHOULD READ THIS THESIS?

For people interested in developing domain-oriented or task-specific systems, this thesis provides insights into what constitutes a "domain," where this shared understanding of the domain comes from, and how this understanding changes over time.

For design theorists, the empirical studies provide insights into macro design processes by characterizing the ongoing interaction between design stakeholders, design tools, and design representations. Specifically, the studies illustrate long-term patterns of change in design representations in response to communicative needs between stakeholder groups and tool affordances and hindrances.

For developers of design support systems, this thesis stresses the notion that "one size does not fit all" when creating design support tools; i.e., that design environments for small teams in dynamic domains have different requirements than those targeted at large design efforts (e.g., designing space shuttles) or mature design domains with well-defined practices and representations (e.g., architecture). With respect to supporting small design teams in dynamic domains, this thesis enumerates guidelines for improving existing design

environments and offers recommendations for the next generation of design support systems. Additionally, developers can reuse the domain construction framework developed in this thesis to assess and compare other models of design support systems not considered here.

For researchers and practitioners in human-computer interaction, this thesis contributes to the ongoing debate concerning the benefits of domain-oriented or task-specific tools versus generic software tools. Furthermore, the findings suggest considerations to be taken into account when creating interfaces and tailoring mechanisms to enable domain practitioners to configure and tailor interoperable sets of software components.

READING GUIDE

This dissertation moves through three main stages: empirical studies, theoretical development and analyses of systems.

Chapter 2 puts forth the central claim of this thesis: that the concept of "domain" is a valuable one, both from a cognitive and a human-computer interaction perspective. Empirical studies are used to illustrate how domains are socially constructed over time and the patterns of change within a particular domain are concretely analyzed. Theoretical arguments are made as to why social domain construction is a desirable phenomena that should be supported by design environments.

Chapter 3 presents the domain construction framework used for analyzing and comparing the flexibility and support provided by design support systems. Processes to be supported consider are organized into those supporting use and those supporting change. The framework considers how different aspects of the underlying computational model affect these various processes.

Chapters 4 and 5 present and analyze the two models considered in this dissertation – the Toolbelt Model and the DODE Model. Each

chapter begins by defining the model under consideration and explaining its significance. The centerpiece of each chapter is a description of the specific design environment instantiating the model developed during this research and an analysis of project experiences from the domain construction perspective.

Chapter 6 compares and contrasts the two models. The objective is to determine how well various aspects of a particular model support the evolutionary and revolutionary patterns of change found in dynamic domain construction. These findings form the basis for Chapter 7, which offers a vision of the next generation of design environments to support designers in innovative and dynamic domains.

Chapter 8 summarizes the key findings of this research and enumerates the core contributions this work has made to the fields of design and computer science.

CHAPTER 2

THE WAY WE DO THINGS AROUND HERE

This chapter examines the nature of design in small, independent design teams working in innovative and dynamic domains. In these types of domains, there are no standard design representations or practices. Rather, it is part of the designers' job to continually create and evolve appropriate representations and practices. In doing so, communities of practice (i.e., a group of design stakeholders) construct their domain over time by defining important domain objects, creating multiple representations for viewing these objects, and establishing relationships between objects and representations.

This process of domain construction is illustrated with three stories drawn from user interface design. User interface design concerns the design, evaluation, and sometimes the actual construction, of the part of a software system that end users see and interact with. User interface design is an ideal example of an innovative and dynamic domain because it is in constant flux due to changes in computational platforms, new interaction devices, and new research findings.

These three stories also illustrate the particular context focused on in this thesis: small, independent design teams working on relatively small projects. They show the pervasiveness of domain construction in a variety of design settings ranging from a university team doing a short, one-off design project to industrial designers working on a continual stream of upgrades to an existing product line. In these different settings, what constitutes the "domain" can assume a wide range of complexity – ranging from a simple set of conventions to elaborate sets of representations and practices.

In this chapter, these design stories are first presented and then analyzed in detail. The first part of the analysis examines what constitutes the domain: design tools, representations, practices, and design languages. Design languages are conventions that design communities construct and evolve to bind together tools, representations, and practices. Finally, the stories are further analyzed to identify the core processes involved in domain construction.

THREE DESIGN STORIES

Every organization has its own rules, operating procedures, and general approaches to doing business. Many people, after changing jobs or institutions, have heard the phrase "that's not the way we do things around here" when trying to apply old practices to their new job context. In this section, I present three stories that illustrate that "the way we do things around here" is not just an organizational phenomenon, but it is also a situation-specific design phenomenon. In each story, what constitutes the "domain" or "the way we do things around here" is situation-specific in that the domain is constructed over time by the particular community of practice to solve their specific design problems.

Story 1: Protocol Analysis

Over the course of a year, myself and three cognitive psychologists conducted a series of experiments to ascertain the effects of using a prototype design environment on the activities of professional designers [9, 10]. As part of the experiment, we asked designers to "think aloud" while using the system to perform a design task. We then transcribed and analyzed the designers' verbal protocols.

While this methodology is a standard technique for evaluating user interfaces, the exact nature of the analysis differs for every system and experimental situation since the analysis depends upon the effects of a specific system on a specific task performed by selected individuals. Thus, analyzing the protocol data is very much a design activity since

the protocol analysts must: (1) design an initial analysis framework, (2) create representations for viewing and filtering large amounts of verbal data, and (3) continually redesign the analysis framework and representations to reflect patterns emerging from the data being analyzed.

During the series of experiments, we collected and analyzed nine protocol sessions. Figure 2.1 illustrates how the representation we used to support the analyses evolved over these nine sessions. For the first analysis, we simply transcribed the verbal protocols into MS WORD, a word processing tool. The representation we used was a simple double-spaced document with some identifying header information. All knowledge of the analysis task resided in our practices, as we painstakingly marked up and annotated the protocol document with colored pens and highlighters and later compared our marked up documents with each other and with the initial analysis framework.

We quickly realized that double spaced documents were not a good representation – there was not enough room for annotations and the verbal data were too spread out to be easily viewed. We went to a single column format. By the fourth analysis (see Figure 2.1), we had standardized header information but, more importantly, we are starting to use formatting to visually distinguish essential data (designer verbalizations) from typically extraneous data (observer comments). The long protocol documents are beginning to be broken into multiple sections corresponding to subtasks in the activity. At this point, our representation has been modified to better support our task but much of the task knowledge still resides in our practices with highlighters and colored pens.

By the ninth analysis, we are working much smarter (see Figure 2.1). We have standardized on a set of named formatting styles that make important things visible (e.g., bold face *subtasks*) and less important things less visible (e.g., small font, italicized *observer/comments*). Key items from our analysis framework are starting to appear as named formatting styles (e.g. underlined *critics*).

Entities such as these did not used to be part of the representation but instead were identified later during the highlighting phase of our practices.

There are several reasons for these improvements. First, our approach improved simply by virtue of doing it over and over again. After every analysis, our understanding of the task deepened and *distinctions, that we had previously only been tacitly aware of, emerged and became explicitly incorporated into our representation*. In natural language, the term "distinctions" refers to articulated objects and qualities that arise through recurrent patterns of breakdown in concernful activity [124]. In this thesis, domain distinctions are articulated objects and qualities that are important for expressing solutions to re-occurring design problems. These domain distinctions are usually articulated by emerging conventions concerning their appearance or, in some cases, by explicit naming. Our representation evolved considerably as we enriched our word processing tool with knowledge, in the form of domain distinctions, important to our specific task.

Second, our awareness of how our tool could support our task changed considerably over time. Besides supporting changing the look of our representation, we realized that the ability our tool gave us to name emerging distinctions was very useful. Naming insured consistency in look and supported our communication process. By this time, the analysis team resided in two different countries and we no longer had the luxury of sitting side-by-side and comparing highlighted, marked up documents to see if we had marked the same things in pink versus green or so on. Now, we could send our partially "marked up" analyses documents back and forth via electronic mail without losing the markings.

Figure 2.1: Protocol analysis story.

Evolution of domain distinctions and representations over time.

I say "partially" marked up because the tool had several limitations that hindered this marking methodology further. First, only paragraph styles could be named. However, a paragraph is too large in granularity for protocol analysis; we need to be able to select single sentences or sets of words and assign them a named style. Second, one big motivation for using named styles rather than just establishing formatting conventions was that we wanted to manipulate the text using the named items to construct different views; i.e., show me only the text labeled *critics* and *anticipations*. However, the word processor's simple show/hide text facility was insufficient for creating the views we desired and we were unable to create them.

Story 2: Multimedia Title Design

This next story discusses experiences of a university team creating an innovative multimedia "title" for presentation at an upcoming conference. The title was an interactive presentation that mixed video clips, still photos, sound, and text to present a vision about the future of programming in the year 2010 [18]. The video clips and the main themes of the title were based on a two day research symposium that had been previously filmed. The design and implementation of the title took approximately four months. The design team consisted of a lead designer from the computer science department, a computer science professor, and two design students from the environmental design department. The details of this story are drawn from a presentation by the lead designer about the design process and subsequent interviews where we examined the tools and representations used during the title's design and production.

The design of the title was an interplay between designing the form and designing the content. Over the course of the project, the team created two primary representations to support this dialectical process that emphasized these different aspects of the design (Figure 2.2).

One representation was a set of six templates illustrating the general form or look of the individual screens. This set of templates was created in Macromind Director, a multimedia authoring tool. One

template (filled in) is shown on the right side of Figure 2.2. Each template consisted of a *vision* statement in green text followed by an *elaboration* on the vision in blue text. There were two main sorts of templates – those with single line visions and those with two line visions. This distinction affected the placement of the *dividing line* between the vision and its elaboration. Each of these two template flavors came in three variations for each of three different *background* colors.

The other representation was a pseudo-hypermedia outline showing the title's overall structure and content; i.e., what individual screens actually said. This outline was created with WORD, a word processing tool. Part of this outline representation is shown on the left side of Figure 2.2. This representation had five main parts. Similar to the template representation, it contained *vision* statements and *elaborations*. However, it also represented different distinctions named *scenarios*, *summaries*, and *presentations*.

The designers had created special looks for several of the distinctions. Visions were represented in a large green font and elaborations were in a black, boldface font. The fonts and colors of these latter two distinctions were chosen to match the representation of the same distinctions in the templates to, in the designer's own words, "help minimize the cognitive load" when moving back and forth between the two representations.

Also, only three of these distinctions (summaries, presentations, and visions) were explicitly named in the representation. These distinctions were not named using the word processor's style naming facility; instead the names were embedded in the text. The reason was that the design team wanted to use the word processor's outline facility for selectively viewing tagged text. To do so, they had to use the word processor's standard text tags of "header 1 - header 4." Thus, a scenario such as "software tools" (Figure 2.2) was tagged with header 1, while elaborations were tagged with header 4. This use of the outline facility was designed to help minimize the cognitive load between the two

design representations. For instance, by choosing not to show header 4 (i.e., collapsing the display so elaborations were not shown), WORD could be configured to display all the visions associated with a particular scenario on one page; this page view of all the visions was identical to their display in the final multimedia title.

As in the protocol analysis story, these designers evolved a set of related domain distinctions to express their design with. The title revolved around five main *scenarios* that had been developed during the research workshop. The original idea behind the *presentations* distinction was to describe how the scenario was presented at the research workshop; however, this idea was never followed through, though it remained in the outline representation. After choosing to interact with a particular scenario in the title, the viewer first hears an audio recording of a *summary* sentence or two. After hearing the summary, the viewer can choose from a set of *visions* concerning the outcome of that particular scenario. The viewer can visit separate screens corresponding to each vision to view further *elaborations* on the vision and to make comments concerning the vision. Like the protocol analysis story, these distinctions were incorporated into the design representations and distinguished by conventions regarding their form and/or explicit names.

However in this design story, *multiple, interrelated design representations were essential for both design construction and communication among all design stakeholders.* The publishing tool was good for designing the look of screens; i.e. color, placement, etc. However, it was not good for easily iterating the design of the content (i.e., the words) during collaborative design sessions since its text editing features were clumsy. Thus designers needed two representations – one targeted at form and one at content – to address these two different aspects of the design. The lead designer reported that he created the hypermedia outline representation to facilitate collaboration with co-designers when primarily designing the content of the title. He said he "needed something to take over there that [they] could all look at and modify... easily."

Figure 2.2: Multimedia story.

Relationships between multiple representations are based on domain distinctions.

The overall design process involved a dialectical interplay between these two representations. The distinctions mediated between the two representations in that they established relationships between the representations' various parts (see Figure 2.2). Vision statements in the outline corresponded with vision statements in the templates. When a vision statement changed in one representation, it also needed to be changed in the other. However, making the change was not a matter of a simple cut-and paste; it was a non-trivial effort involving a complex sequence of steps using multiple special effects tools. These well-defined steps transformed basic, unformatted text from the word processor into formatted text on a special paper-textured background in the multimedia publishing tool. Since there were 23 of these individual vision screens in the final title, this was quite a laborious effort. The lead designer reported that, once transformed, only glaring mistakes such as a spelling errors motivated them to iterate through this process again and, as a consequence, small refinements to the content were not made.

In this project, the designers were able to enrich individual tools with some knowledge of important domain distinctions. However, the designers' tools did not support establishing or maintaining relationships *between* parts of the two representations created with different tools. It was the designers themselves; i.e. their practices, that took care of establishing and maintaining correspondences between domain distinctions in the different representations.

Story 3: Voice Dialog Design

The story presented here is drawn from a three year collaboration between user interface designers at US WEST Advanced Technologies (a regional phone company) and researchers at the University of Colorado. This analysis is excerpted from a case study of these designers' practices [116, 119] and resulted from a combination of collected qualitative data [127] including workplace observations, field notes, analyses of existing design tools and representations, videotape analyses, and open-ended interviews with seven members of the

design group. These members included the group's manager, lead designers, other designers, simulation builders, and student interns.

The task is the design of voice dialog applications; i.e., software applications with phone-based user interfaces. These interfaces consist of a series of voice-prompted menus requesting the user to perform certain actions; e.g., “to listen to your messages, press 1.” The caller issues commands by pressing touch-tone buttons on the telephone keypad and the system responds with appropriate voice phrases and prompts. Typical applications are voice information systems and voice messaging systems. Designing in this domain means specifying the interface for an application at a detailed level. During this study, most of the interfaces designed involved modifications and additions to an existing voice messaging product.

There are two main facets to the designer's job: constructing the design and communicating the evolving design to other stakeholders such as marketing and the vendor organization. As one designer noted, "the critical problem is communicating the design to other people in a way that doesn't require a lot of specialized training [on their part]." As in the multimedia story, the designers have created multiple design representations that emphasize different aspects of the design. In this case, each of the four major representations – flow charts, tables, test plans, simulations – is tailored to the special needs of each of the major stakeholder groups.

Flow charts are the primary design representation and are constructed to communicate the essential aspects of the interface such as spoken prompts, menus, and control flow to the marketing, customer support and vendor organizations. Additional detailed design information is presented in a separate table representation that is constructed primarily for the vendor organization. Test plans are semi-structured text documents that detail the actions to be performed and the audio output to be heard along each path through the interface; this representation is constructed for testers who are hired to test all features of the final product. Simulations are functioning prototypes of

critical or contentious parts of the interface and are constructed for the marketing organization and for usability testing.

However, as Figure 2.3a-d shows, both the representations and the distinctions these representations depict have changed considerably over time. At the beginning of the period of study, designs were represented using primarily textual specification documents (Figure 2.3a). These textual specifications distinguished between *voice menus* (using document sections) and *phrases* and *messages* (using formatting conventions). Important information such as dynamic conditions and help messages were buried in the middle of paragraphs.

Two types of system responses are distinguished: phrases (all caps) and messages (italics). Possible user actions are described in indented text. Control flow is described using natural language.

5.3 Personal Options Menu

This section describes the Personal Options menu in the new mailbox type.

> ***(O)*** *"Personal options.*
> *To change your security code, press 1.*
> *TO CHANGE YOUR RECORDED NAME, PRESS 2.*
> *To disconnect, press *. "*

This menu matches the standard Personal Options menu seen in previous mailbox types. A new feature is being added to the Personal Options menu. This is a new option that will allow the user to change their "recorded name." This requirement is detailed in section 5.3.2.

All phrases are interruptible.

If the user does not respond, the system plays *"are you still there?"* and repeats the menu.

If the user requests help, the menu is repeated.

If the user presses *, the system backs out to the Main Menu.

If the user presses 3-9 or #, the system plays *"you have pressed an incorrect key"* and repeats the menu.

If the user presses 1, the system responds as described in section 5.3.1.

If the user presses 2, the system responds as described in section 5.3.2.

Note: The numbers in the "Go To" section of the chart represent section numbers.

5.3.1 Security Code Entry

When the user presses 1 at the Personal Options menu, the system plays **(P)** *"Choose a security code that is easy for you to remember. Enter the new security code now, then press #."*

If the user enters too few digits, the system plays "SECURITY CODE MUST CONTAIN AT LEAST 4 DIGITS" and repeats the prompt.

Figure 2.3a: Supremacy of textual specifications (pre-1991).

These tables depict possible actions after phrases or voice menus request users to do something. Phrases are assigned identifiers which are bold letter/number combinations. The identifiers are interspersed into the text to tie it to the tables.

ENTER YOUR SECURITY CODE

Phrase (P): ***"Choose a security code that is easy for you to remember. Enter the new security code now, then press pound."***

ACTION	EXPECTED RESPONSE	GOTO
timeout	"Are you still there?"	**(P)**
too few	"Security code must contain at least ..."	**(P)**
too many	"Security code may not exceed ..."	**(P)**
digits, *	"Reenter your security code, then press #."	**(P)**
press 0	"Your current security code is (DIGITS)."	**(P)**
press *	"Command cancelled. Previous code saved."	**(O)**
press #	"Enter the new security code now ..."	**(P)**
digits	"For your security code ..."	**(Q)**
digits, #	"For your security code ..."	**(Q)**

PERSONAL OPTIONS MENU:

Phrase (O): ***"Personal Options. To change your security code, press 1.*** **TO CHANGE YOUR RECORDED NAME, PRESS 2.** ***To disconnect, press *.***

ACTION	EXPECTED RESPONSE	GOTO
timeout	"Are you still there?"	**(O)**
press 1	"Security code..."	**5.3.1**
press 2	"Recorded name..."	**5.3.2**
press 3	"You have pressed an incorrect key."	**(O)**
press 4	"You have pressed an incorrect key."	**(O)**
press 5	"You have pressed an incorrect key."	**(O)**
press 6	"You have pressed an incorrect key."	**(O)**
press 7	"You have pressed an incorrect key."	**(O)**
press 8	"You have pressed an incorrect key."	**(O)**
press 9	"You have pressed an incorrect key."	**(O)**
press 0	"Personal Options..."	**(O)**
press *	"Main Menu..."	**(I)**
press #	"You have pressed an incorrect key."	**(O)**

Figure 2.3b: Augmentation of textual representation with simple flow charts and tables (pre-1991).

The phrase tables have evolved into two distinctions: prompts and voice menus. Messages are still present. A new distinction, decision points, are now represented in the flow charts. Control flow is depicted graphically with links. Every object in the flow chart has an identifier that ties it to a corresponding table representation.

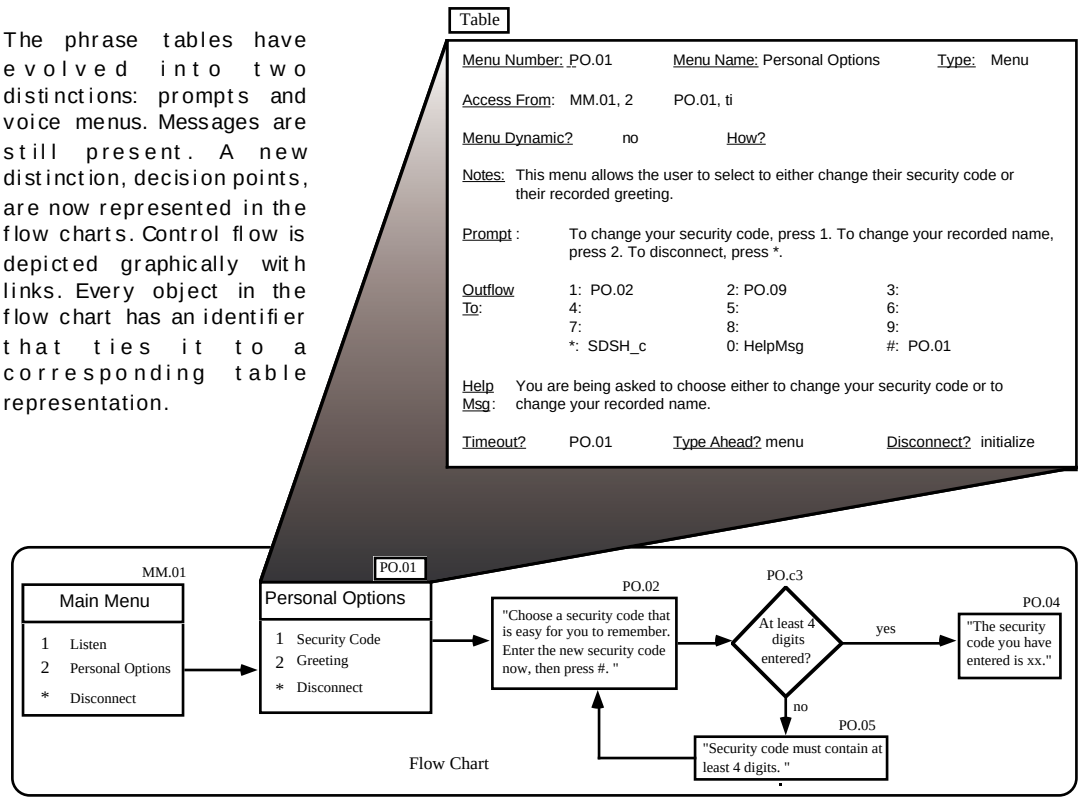

Figure 2.3c: Introduction of complex flow charts and tables as the primary representational system (1991).

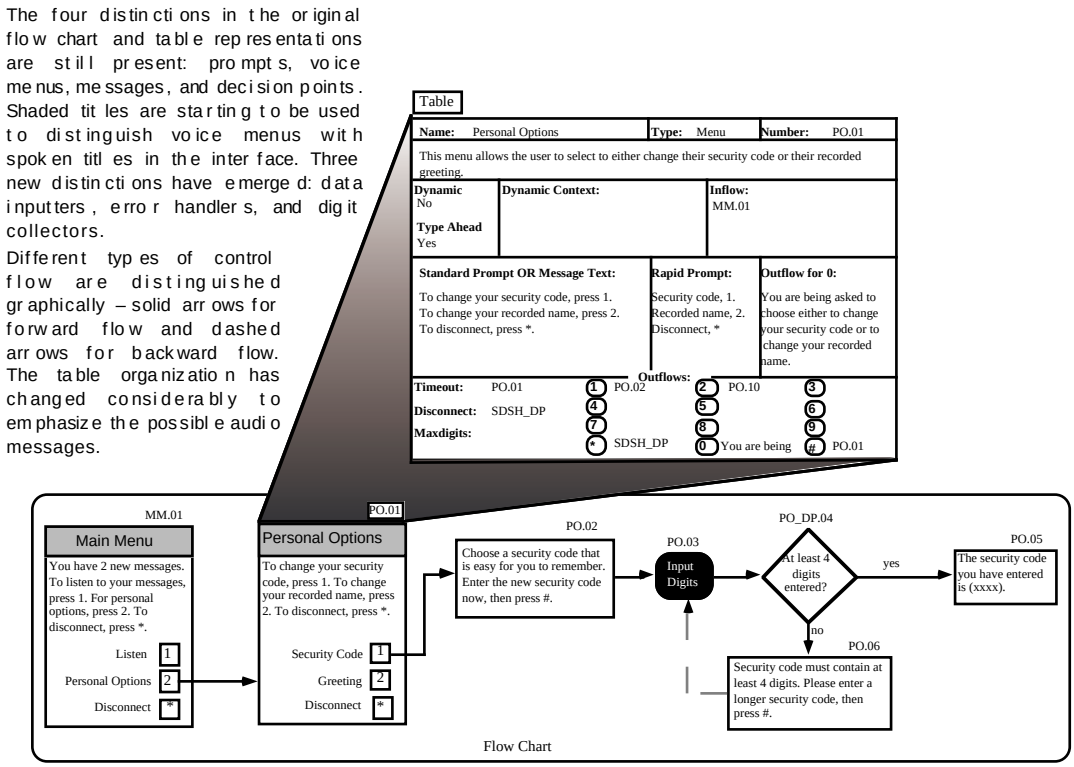

Figure 2.3d: Flow charts and tables become the preferred representational system (1992).

Later, these textual specifications were augmented with simple tables and flow charts at the request of the vendor organization (Figure 2.3b). The simple flow charts did not contain the full text of all audio messages and phrases in the interface. The tables augmented the textual description of prompts and menus by depicting all possible actions and responses. The relationships between entries in the table and parts of the textual document were indicated using bold-face identifiers based on a *letter* or *letter.number* naming convention; e.g., B.1, where the first phrase in the document was named "A" and cycled through the alphabet in sequence from there.

In 1991, one designer was asked to create one of the most complex applications to date. He decided it was time for a whole new approach. The textual specifications were getting so large and complex that few people bothered to read them. Those who did had trouble understanding them. This designer established a personal convention

of using flow charts and a new, complex table as the primary design representations (Figure 2.3c). His flow charts contained four different domain distinctions (*types*): *voice menus*, *prompts*, *messages*, and *decisions*. Each node in the flow chart had a corresponding entry in the table representation. Correspondence was indicated using a new *two letter.two digit number* naming convention where the two letters indicate the design subcomponent; e.g., PO.01 indicates the first node in the personal options design subcomponent. The tables contained design information not found in the flow chart (such as *dynamic* conditions and *help* messages) and redundant information (such as *access from* and *outflows* which are represented using links in the flow chart).

The two representational systems – textual versus flows and tables – coexisted for about one year. Other designers were reluctant to switch for two reasons. First, several had long-term relationships with their vendors and marketers and were reluctant to switch representations on them. Second, constructing the flow charts was tedious and time-consuming. Even simple design modifications were cumbersome since, using existing tools, links (arrows) did not track the movement of nodes (boxes) and had to be manually redrawn.

By mid-1992, several events led to the emergence of flows and tables as the primary design representations. First, new versions of flow charting tools made it much easier to construct the flow chart representation. Second, new designers were hired to embark on a long series of enhancements to an existing product. These new designers readily adopted the flow and table representations.

The new designers began to elaborate on the flow and table representations (Figure 2.3d). New domain distinctions emerged such as *inputters*, *error handlers*, and *digit collectors*. Some designers began to use patterns such as solid versus dashed lines to differentiate between *different types of flow* control. The layout of the table representation has been redone to emphasize the different kinds of voice output (*standard prompts*, *rapid prompts*, and help prompts

(*outflow for 0*). The vendor liked the formality of the table representation and suggested ways to formalize the representation of *dynamic context* using pseudo-code.

It would be a mistake to assume that these current flow chart and table representations contain all relevant domain distinctions and relationships; i.e. that these are the final "picture" and will never change. There are many aspects of this domain that are still being developed and will be furthered refined as the result of ongoing design activities. Also, further signs of major change are looming on the horizon. One designer has started working with a new vendor who (so far) has not required the table representation. In another recent design, the marketers were dissatisfied with the increasing complexity of the flow charts; they think simplification might be in order.

As shown in Figure 2.3a-d, *the design representations and the distinctions these representations depict have continually evolved throughout the three years that this study encompasses. These evolutionary patterns of change included both periodic upheavals and continual incremental elaboration and refinement.* The voice dialog domain, as practiced by this design community, continually evolved as existing distinctions were refined and new distinctions emerged and were incorporated into existing design representations. However, the domain also underwent periodic upheavals as new representational systems were introduced, either to augment existing ones or to replace systems that no longer met design needs.

WHAT IS A DOMAIN?

The previous three stories provided a flavor for how design communities construct their "domain" over time. However, the question remains, what is a domain? Central constituents of the domain are the *tools* and *practices* used by a design community to create *products* pertinent to solving their design problems. But tools, practices, and products do not by themselves constitute a domain. The essential aspect of a domain is the shared understanding between

domain practitioners that enables a community to use tools in a practiced way to create products in a consistent manner. For instance, in the voice dialog design story, off-the-shelf software tools such as flow charting packages and databases were used in a constrained way to create design representations or products (e.g., flow charts and tables) with a very specific content and look.

This shared understanding binds tools, practices, and products into a domain. I call this shared understanding a *"design language"* (see Figure 2.4). Over time, design languages take on a tangible form: in the stories we observed how domain distinctions gradually emerged through practice and became incorporated into progressively elaborate and standardized design representations. Design languages consist of a vocabulary of domain distinctions, one or more representational systems highlighting important aspects of and relationships between these distinctions, and evaluative knowledge for judging or assessing the qualities of a design product.

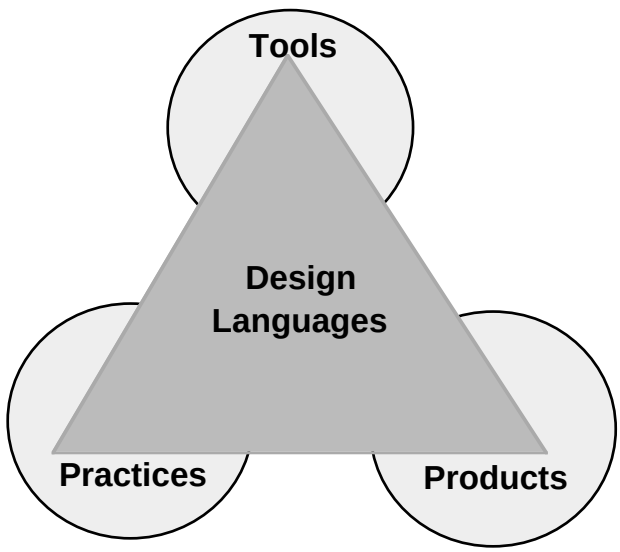

Figure 2.4: Relationship between design languages, products, tools, and practices.

Tools, practices, and products can take a variety of forms as indicated by their surrounding circles. It is the design language (shaded triangle) that binds together the tools, practices, and products into a domain.

Vocabularies of domain distinctions are the core constituent of any design language. In natural language, "distinctions" are articulated objects and qualities that arise through recurrent patterns of breakdown in concernful activity [124]. Likewise, in design languages, domain distinctions are articulated objects and qualities that are important for expressing solutions to re-occurring design problems. In the voice dialog design story, designers articulated distinctions such as voice menus, prompts, and messages. In the multimedia title story, designers articulated distinctions such as visions, elaborations, and scenarios.

Representational systems are visual formalisms that have been tailored to support the work practices of a specific design community. According to Nardi, visual formalisms are diagrammatic displays with well-defined semantics for expressing relations and these semantics are broadly applicable across many domains [79]. She cites generic table and flow chart representations as examples of visual formalisms. The visual formalisms observed in this work were continually tailored by the designers themselves over extensive periods of time. They were tailored such that their look and structure were modified to make important domain distinctions and relationships visually apparent. For instance, the notion of a hypermedia outline is an example of a generic visual formalism; but the specific hypermedia outline used by the multimedia designers is an example of a tailored representational system. These designers extensively modified the outline to reflect their domain-specific vocabulary (e.g., scenarios, visions, elaborations, etc.) and to emphasize the relationships between the hypermedia outline and the template representation.

The third constituent of design languages is articulated *evaluative knowledge* concerning what constitutes a good design. Evaluative knowledge are criteria and constraints that designers use to judge the quality and deficiencies of a given design with respect to certain goals [8]. As such, this knowledge is not tied to a particular representational system but rather a good system will make it easier for designers to evaluate their design. Neither the protocol analysts or the multimedia title designers had explicitly articulated any heuristics. However, the

voice dialog designers had articulated numerous design heuristics, namely in the form of lengthy user interface guideline documents. These documents contained heuristics such as "voice menus should not contain more than four prompted menu items" or "menu commands should be in ascending, numerical order." In their practices, creating designs consistent with the existing product line is an important design heuristic. However, what it means for designs to be "consistent" is not explicitly articulated.

Over time, these three design communities all evolved increasingly well-defined design languages that supported their specific design practices. Design languages are well-defined when practitioners use them to create products with a reproducible, consistent look and structure that are reliably interpretable by members of the design community. In the following section, we will examine the processes by which well-defined design languages come into being.

THE EVOLUTION OF A DOMAIN

In the previous section, I argued that well-defined design languages are the crucial element binding tools, practices, and products of a specific design community into a "domain". *Domain construction* refers to the processes by which well-defined design languages come into being. In this section, we will look across the three design stories to derive a better understanding of these processes. First, we will examine the processes by which tacit understandings evolve into articulated domain distinctions. Second, we will look at how and why design communities evolve generic visual formalisms into multiple, interrelated representational systems. Finally, we will look at how changes in tools, products, and practices lead to both evolutionary and revolutionary changes in existing design languages.

From Tacit Understandings to Articulated Domain Distinctions

Polanyi [88] noted that professionals know more than they can say; i.e., that much professional knowledge is tacit in nature. Schoen, in his studies of professional practice, also argued that professional action is

highly tacit and situated [103]. However, Schoen observed that occasionally during the course of design, situated action breaks down and designers engage in periods of "reflection-in-action" during which new explicit knowledge may be created or existing knowledge applied in novel ways.

Stahl carried these arguments further in his philosophical analysis of the role of interpretation in design [111]. Stahl argued that a central part of the design process is the successive transformation of information from tacit preunderstandings to codified knowledge. Specifically, he claims that during the course of design, knowledge may go through the following transformations:

(1) Designers begin with a tacit feeling that they have uncovered a design problem or opportunity.

(2) This tacit preunderstanding is interpreted by the designers in a specific way, resulting in an explicit understanding of the problem or opportunity.

(3) Designers then make assertions about the design issue; these externalized expressions in language enable the issue to be discussed and communicated.

(4) To preserve their interpretation of the issue, designers predicate this knowledge using some semi-formal or formal method such as IBIS resulting in codified design knowledge.

Stahl further argues that design knowledge existing at the tacit preunderstanding and the explicit understanding stages is in the realm of individual human understanding. It cannot be shared by a design community until it is transformed to externalized expressions or codified knowledge.

In the two stories for which we have process data, the protocol analysis and the voice dialog design story, we observed the basics of Stahl's interpretative process. However, domain distinctions were the

primary form of design knowledge observed to be externalized and codified. In both stories, individual reflection-in-action and pressures to communicate effectively with other design stakeholders appeared to drive the articulation of domain distinctions.

In the beginnings of the protocol analysis and voice dialog stories, the design representations consisted mainly of ill-defined, natural language text documents that contained very few, explicitly distinguished domain distinctions. The early voice dialog textual specifications only had three distinctions: messages, phrases, and voice menus. The first protocol analysis representation did not contain any domain distinctions.

Some distinctions found in later representations are entirely missing from early representations. Other domain distinctions found in later representations were only tacitly represented in early representations in the sense that important objects and relationships were often buried, unmarked and even unnoticed, in the middle of paragraphs. In the voice dialog design case, caller input, such as the system pausing for a fixed period of time while the caller enters a 4 digit security code, is designated by a special inputter object in later design representations. However, this information is only implicitly represented in earlier representations and must be pieced together by reading surrounding phrases and error messages. This lack of explicitly distinguished domain distinctions reflected the tacit nature of the designers' existing preunderstandings.

As the designers repeatedly practiced their craft, domain distinctions emerged in the sense that designers began to explicitly signal their presence using special notations. Figures 2.1 and 2.5 illustrate the distinctions that emerged during the protocol analysis and voice dialog design stories respectively. Towards the end of the stories, all three design groups had developed vocabularies consisting of several articulated domain distinctions.

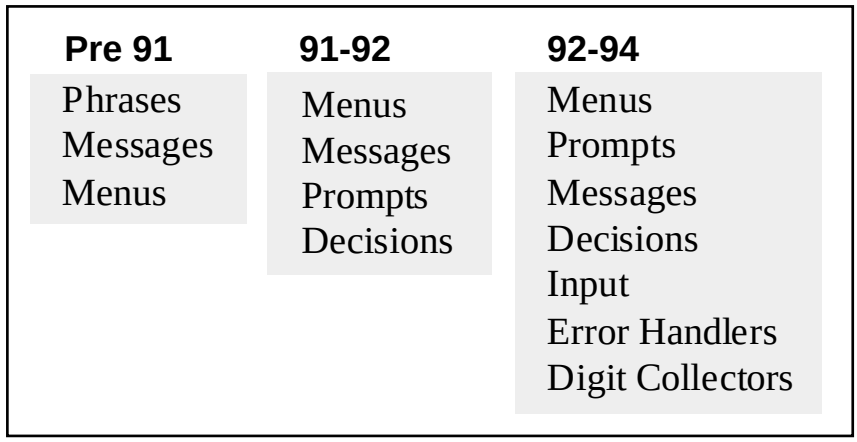

Figure 2.5: Evolution of domain distinctions in the voice dialog community from 1991 through 1994.

In Stahl's analysis, the primary mode of externalized expression is *naming*; i.e., assigning a meaningful identifier to a design concept. His transformation stages indicate that naming is a precondition to codification of knowledge. Certainly naming of domain distinctions was an important process in our design stories. However, it was not necessarily a precondition to codification. An equally important process we observed that combined aspects of externalized expression with codification was *graphical refinement*, a process whereby designers incrementally elaborate on the look of a distinction within the context of a particular representational system. In our design stories, many times graphical refinement occurred before a distinction was named and in some cases, the distinction being elaborated on was never explicitly named. For instance as shown in Figure 2.1, the protocol analysts had performed many graphical refinements by the fourth analysis but did not begin to name these distinctions until later analyses.

The protocol analysis story provides a good example of graphical refinement (Figure 2.1). By the fourth analysis, the analysts are articulating domain distinctions graphically using formatting features such as bold, italics, and different size fonts. At that point, none of the distinctions distinguished graphically are explicitly named. By the ninth analysis, many of those distinctions are named but new unnamed distinctions are also beginning to emerge graphically.

Thus, in all three design stories, domain distinctions were *articulated* by being named and being given a specific graphical look. Graphic looks can serve many purposes. At the shallowest level, they serve to visually distinguish between the different domain distinctions. At a deeper level, they also convey crucial information about the particular distinction such as its components, its function or behavior, or its relationship to other distinctions. In the multimedia case, color and fonts were used to indicate relationships between distinctions (e.g., elaborations and visions) across different design representations; specifically to reinforce the reminder that these distinctions should be kept consistent across representations. The voice dialog designers had the most developed distinctions in terms of using graphic looks to convey detailed information concerning function and components. For instance, the look of voice menus in the flow chart indicate both modality and component information. Figure 2.6 shows how the graphic look of voice menus in the flow chart representation evolved over time to show more behavior and component information.

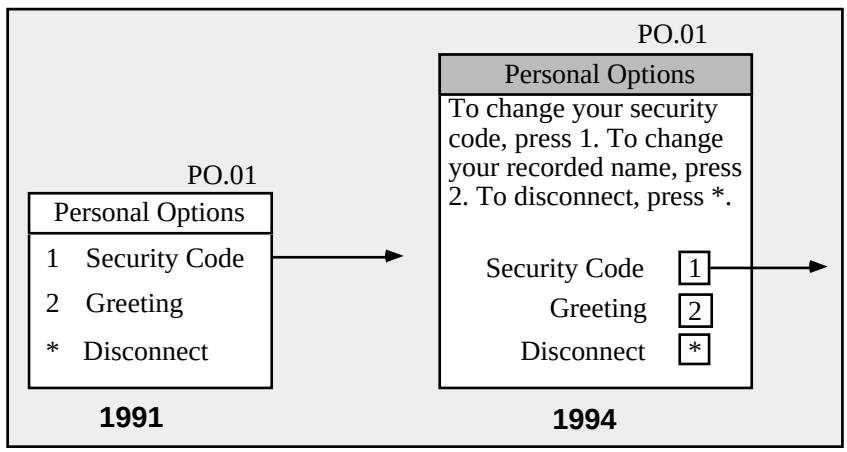

Figure 2.6: Evolution of the voice menu representation.

When flow charts were first introduced in 1991, voice menus had a very simplistic look and content. By 1994, voice menus have evolved an elaborate look and content: shading indicates voiced menu titles, miniature touch tone buttons emphasize legal key presses, and the menu prompt is shown.

From Visual Formalisms to Interrelated Representational Systems

The three stories illustrate how representational systems are central to the design languages used by the design communities and how a large part of the communities' activities revolved around the creation and continual modification of their representational systems. These observations correspond with empirical findings in other design domains. Schoen's analysis of professional designers such as architects revealed how a major part of a designer's job is to create and evolve external representations of the design being constructed [103]. In software design, an important activity for expert designers is figuring out what representational systems to use or even creating new ones if necessary [15].

In all three stories, initial generic visual formalisms such as flow charts, tables, or text documents were continually tailored by designers to better support their specific design needs. In the previous section, we saw that an important part of this tailoring process is the articulation of domain distinctions in the context of design representations. Over time, the representations in all three design stories contained progressively more domain distinctions depicted with greater levels of detail.

A second aspect of the tailoring process is the refinement of other aspects of the representational systems such as layout and other markings that are not part of the design language's vocabulary. In the protocol analysis case, the analysts' refined the basic protocol text representation by parceling the protocol text into more and more subareas with distinct header information (see Figure 2.1). In the voice dialog design story, the initial flow charts were difficult to read due to the large number of arrows connecting nodes. Designers began to experiment with various refinements to reduce the number of arrows. Specifically, some designers only showed backwards flow arcs when the flow behaved in a non-standard way. All designers began to use "go to" labels to replace backwards arcs that would cross large areas of the diagrams. Additionally, while the distinctions contained in the voice

dialog table representation stayed about the same, the layout of the table representation changed considerably over time to increase the emphasis on the various voice recordings associated with each item (see Figures 2.3 C-D). These refinements are analogous to the development of secondary notations in CAD design observed by Petre [86]. According to Petre, secondary notations are valuable layout clues that are not part of the formal notation that exhibit relationships and structures that would otherwise by less accessible [86]. In our stories, many layout practices that began as informal secondary notations eventually were incorporated as explicit entities into the representational systems.

A third aspect of the tailoring process is the creation of multiple, interrelated representational systems to support design construction and communication. In the multimedia and voice dialog design stories, multiple external representations were essential to the overall design processes. The multimedia designers constructed two representations – a hypermedia outline and presentation templates. The voice dialog designers constructed four major representations – flow charts, tables, simulations, and test plans.

Multiple design representations are required for several purposes. First, representations at different levels of abstraction support design construction by enabling designers to engage in opportunistic design. Opportunistic design refers to how designers interleave decisions at various levels of abstraction during solution decomposition as a consequence of the ill-structuredness of problems in the early stages of design [48]. The two representations used by the multimedia designers illustrate representations supporting opportunistic design. The hypermedia representation was targeted at higher-level design issues such as planning the overall structure and content of the title. The presentation templates supported designing the more detailed form or look of the title. The lead designer reported that these two representations were used continually throughout the project's design process.

Second, design activities in the workplace usually involve several different stakeholders from a variety of backgrounds [101]. Often, designers must construct several external representations to facilitate communication and collaboration with each stakeholder group [20]. Several researchers claim that this process of developing shared artifacts such as external design representations is not only desirable, but fundamental to the design process. These researchers note that design is really an iterative process of cooperative learning between stakeholders and such shared representations are crucial for providing a frame of reference in which to ground successive learning cycles [60, 84].

In our three design stories, we saw that improving communication with other design stakeholders was a major factor motivating the creation of and changes in representational systems. The lead multimedia designer reported that he created the hypermedia outline because he needed a representation to take over to his colleagues in environmental design that facilitated iterative, collaborative design.

The voice dialog designers provide an extreme example of creating representational systems primarily to facilitate communication with other stakeholders. Each of their four major representations is targeted at the needs of a specific stakeholder group. The flow charts are the primary representation and are used by all design stakeholders, but their particular content is inspired by the needs of the marketing group and customer support. Both the early and the later, more complex table representations were created to satisfy the vendor organization. The simulation representation, which is accessible via a standard telephone, facilitates long-distance collaboration with the marketing group which resides in a different state. And the test plans are constructed primarily for testers which are often contractually hired to verify that the delivered product meets the design specification.

Though each representation emphasizes different aspects of the design, many domain distinctions are shared by several representations. Figure 2.2 shows some of the distinctions shared by the

hypermedia and presentation representations in the multimedia project. This sharing of distinctions results in many complex relationships and dependencies between different design representations. Figure 2.7 shows the different types of relationships between the flow chart and table representations in the voice dialog design case.

Relationships between representational systems fall into four main categories. *Correspondence* relationships indicate which parts of different design products are related; no attempt is made to relate to the content of the distinctions. In Figure 2.7, the entire Personal Options menu representation corresponds to the personal options table entry. *Equality* relationships indicate which distinctions and their subparts should have identical content. In Figure 2.7, the prompt in the flow chart menu and the standard prompt in the table should be identical. *Transformation* relationships indicate that content in one representation can be mechanically derived from content in the related representation without recourse to domain-specific knowledge. For instance, the outflows in the table representation can be derived through the transformation process of noting the identifiers of nodes connected by outgoing links (Figure 2.7). *Heuristic* relationships refer to how content in one representation is related to content in another representation through the application of domain-specific knowledge. The rapid prompt in the table representation in heuristically related to the prompt in the flow chart representation.

Establishing, *evolving*, and *maintaining* relationships between representations is a large part of the design process. Some relationships are established when the representational systems are initially created. For instance, the basic relationships between the flow chart and the table representation were established at their creation in 1991 (see Figure 2.3c).

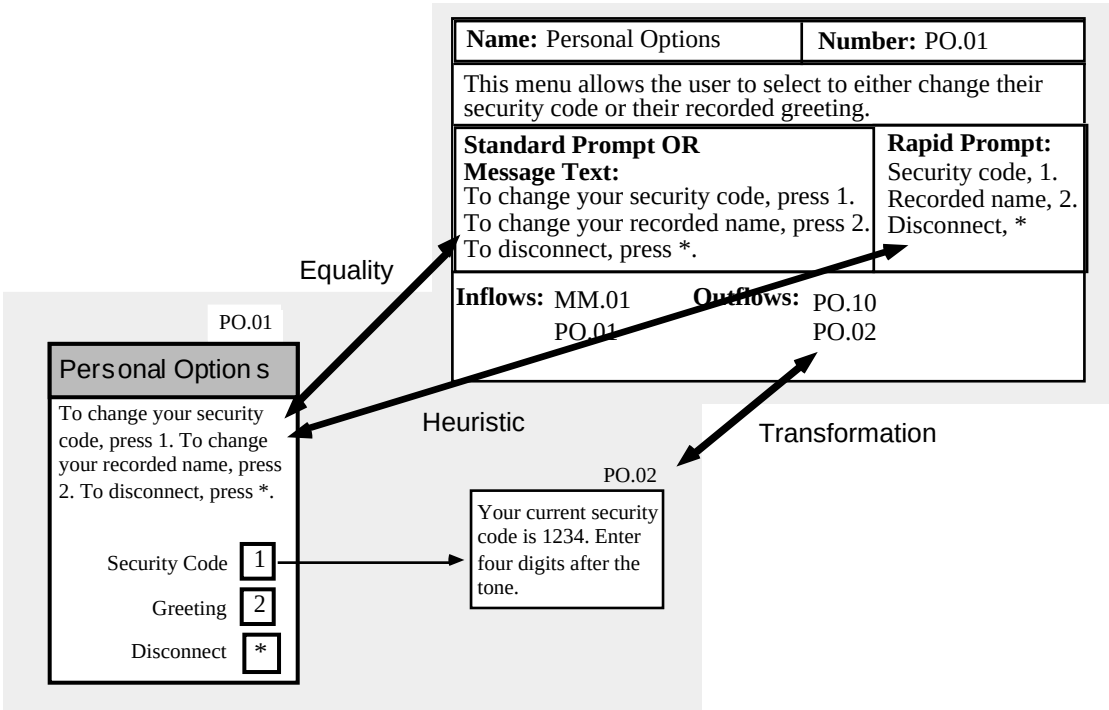

Figure 2.7: Four types of relationships.

The entire Personal Options Menu in the flow chart *corresponds* to the related entry in the table representation. In addition, there are *equality*, *transformation*, and *heuristic* relationships between different distinctions in the flow chart and table representations. See text for more details.

Many relationships evolve as representational systems co-evolve to better support design construction and communication processes. The correspondence relationship between the menu prompts in the flow chart and the table representation could obviously not exist until menu prompts were added to the flow chart representation in 1992 (see Figure 2.3d). In the multimedia case, the steps required to transform visions and elaborations in the hypermedia outline into their final appearance were not completely known until the design of the form was finalized towards the end of the project. Representational systems also co-evolve to emphasize or make apparent important relationships. A good example of this phenomena is how the look of visions and elaborations in the hypermedia outline were made to match the same distinctions in the presentation template to lessen the multimedia designers' cognitive load.

When engaging in opportunistic and iterative design, maintaining relationships across representations is a large part of the overall design process. As designers modify one representation, related distinctions in all other representations must be modified to ensure design consistency. As an example from the voice dialog case, imagine that usability testing reveals that most callers want to hear their existing security code before they change it. The designer decides to add a new confirmation message between the Personal Options menu (PO.01) and the change security code prompt (PO.02) (see Figure 2.3d). This requires the designer to adjust all subsequent identifiers to make room for the new message. Additionally, each table corresponding to a changed entity in the flow chart must also be updated with the new identifier and all the inflow /outflow lists must be reworked. If the designer decides that this change should undergo usability testing, the design simulation must also be modified. Finally, any test plans traversing the path containing the new message must then be updated.

In summary, tailoring generic visual formalisms to create representational systems is an important domain construction process. Tailoring activities include the articulation of domain distinctions, the refinement of secondary notations and other layout considerations, and the development of multiple, interrelated representational systems to facilitate design construction and communication with other stakeholders in the design community.

From Ill-defined to Well-defined Design Languages

The design languages used by the three communities went through a maturation process as they progressed from being ill-defined to well-defined. In the voice dialog case, the original textual specifications were ill-defined because important aspects of the domain were not made explicit by the representation. Key interface features were buried in the middle of paragraphs and this resulted in design errors and communication breakdowns. As designers recognized common breakdowns in the design process, they created objects and representations to overcome these breakdowns. The outcome was a

progression towards well-defined design languages that made explicit significant domain objects and their relationships. These well-defined design languages were more consistently reproducible and interpretable by the design community.

In all three stories, as practitioners worked, they constructed their domain by refining existing distinctions and representations and by creating new distinctions and representations. Sometimes these new representational systems augmented existing ones; other times, they replaced existing systems that no longer met design needs. Thus, the progression towards well-defined design languages was not a simple, accretionary process where more and more distinctions were articulated and more representational systems were created. Rather, periods of *elaboration* on existing design languages were punctuated with periods of *revolutionary upheaval* as existing design languages were replaced. We observed periods of elaboration and upheaval in both stories for which we have process data. The protocol analysts quickly replaced their initial full page, single column representational system when they realized it did not support their design needs; this was followed by a long period of elaboration on the new, two column representation (see Figure 2.1). The voice dialog designers went through a radical change in their design language when textual specifications were replaced with flow charts and tables. This change was followed by a long period of elaboration as the content and look of the flow charts and tables were refined (see Figure 2.8).

The concept of genres of organizational communication [126] provides insights into these elaboration and upheaval episodes. According to this concept, the nature and role of communication is always evolving as individual actors interact with social institutions over time. Genres of organizational communication are widely recognized types of discourse (e.g., letters, memoranda, audit reports) that are characterized by structural, linguistic, and substantive conventions or rules. A genre has common subjects (i.e., content) and common formal features (e.g., graphic look and layout) that make it recognizable and interpretable by others; these shared conventions and

rules are the by-products of a history of negotiations among social actors in a community. Genres emerge within a particular context and are reinforced; i.e., their rules for consistent reproduction are strengthened, as a situation re-occurs over time.

Design products can be considered a form of organizational communication. Design languages are the genre rules that define a product's content and look and enable the product's consistent production and interpretation. These genre rules arose from negotiations over effective communication within the design communities. A good example is how one voice dialog designer pioneered the flow chart and table representations because he feared other stakeholders were no longer reading the textual specifications. Initially, the marketing group was skeptical concerning the merits of the new representations. However, after a period of continual modifications, often at the marketers' behest, the group now prefers the new representations. Sometimes other stakeholders suggested changes to products, such as the request from the vendor organization to augment the textual specifications with phrase tables.

Yates [126] observed the basic processes of maintenance, elaboration, and modification in the evolution of "memos" as a communicative genre over the last one hundred years. Maintenance occurs when individuals enact genres using the rules of substance and form without alteration. Elaboration occurs when individuals consistently but slightly adapt genre rules to reflect new conditions. Modification of genres occurs when individuals depart significantly and persistently from existing rules, such as when prose reports are replaced by tabular numeric reports in organizations.

We observed all three processes in our stories. Figure 2.8 tracks the genres used by the voice dialog designers over a three year period. The primary difference between our design stories and Yates' findings seems to be the rate of change. In Yates study of memos, genre maintenance was the dominant process. In innovative and dynamic design domains, genres (i.e., design languages) evolve at a faster rate

and therefore elaboration and modification (i.e., revolutionary upheaval) are the dominant processes. In design languages, elaboration processes consists of vocabulary articulation via naming and graphical refinement and the establishment of new relationships between representational systems. Modification processes involve creating new representational systems to either augment, co-exist with, or replace existing systems.

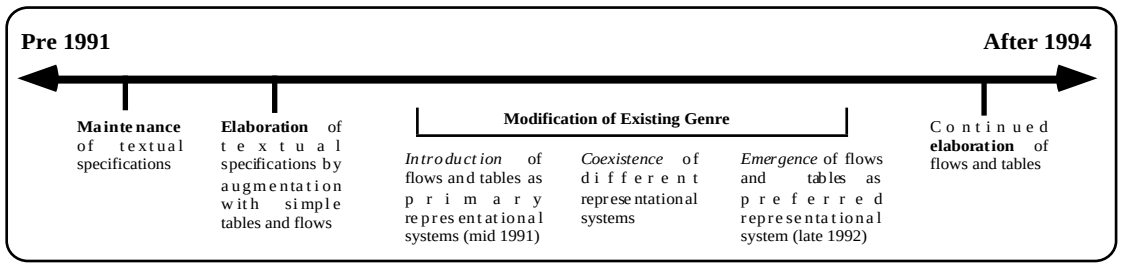

Figure 2.8: Patterns of change in design languages.

The design language used by the voice dialog design community went through cycles of elaboration and modification over a three year period from 1991 to 1994.

SUMMARY

In summary, the three design stories showed how:

- Designers gradually evolve specialized representations tailored to their particular needs by continually refining their form and their content.
- Design communities created multiple, interrelated representations that emphasized different aspects of the design.
- Periods of elaboration on existing representations were punctuated with periods of revolutionary upheaval as existing representations that no longer suited practices were radically modified or replaced.

I argued that domains are not static entities that objectively exist, but instead are dynamic entities that are constructed over time by design communities. Domains consists of design tools, products, and practices bound together by design languages. Domain construction

refers to the processes, illustrated in the stories, by which well-defined design languages come into being.

In this chapter, I concentrated on the complex relationships between design products, design languages and design stakeholders. In design communities, individual communicative actions (e.g., construction of design products) both shape and are shaped by socially constructed genres (e.g., design languages). Such recursive interactions between human actors and social institutions are referred to as adaptive structuration processes [83, 126].

However, another factor influencing the structuration process is technology. According to adaptive structuration theory: "humans create technology with particular physical features, and in their encounters with those technologies, behave in ways that are partly a function of pre-existing patterns (i.e., genres) and partly a function of the opportunities and constraints made manifest in the technology," page 156 [35]. This co-adaptive phenomena is also documented in Mackay's studies of the introduction of customizable tools into work groups. She observed people to not only adapt tools to better support their practices but also to adapt their practices to make more effective use of the tools [68]. Thus, what ultimately constitutes a domain is the result of a co-evolutionary process between design tools, products, practices, and design languages. The remainder of this thesis will focus on tools, exploring how two specific models of computational tools may hinder or support domain construction processes.

CHAPTER 3

DOMAIN CONSTRUCTION FRAMEWORK

In Chapter 1, I argued that small, independent design teams working in dynamic domains are increasingly important segments of the economy that could benefit from computational support. However, this raised the question of how to strike a balance between supporting *use* and supporting *change*; i.e., how much and what types of flexibility do design environments need?

Chapter 2 investigated the flexibility required. Analyses of three design communities characterized the patterns of change in design tools, practices and products. These analyses showed how design communities gradually construct their domain by defining and refining domain objects, creating and evolving multiple representations, and establishing complex relationships between objects and representations. This process was labeled *domain construction*.

This chapter analyzes in detail the subprocesses involved in domain construction. These analyses help to deduce requirements indicating the *ideal* functionality that design environments should provide in order to strike a balance between supporting use and supporting change. However, the ability of environments to support these processes depends upon several aspects of the overall computational model the environment is based on, such as the underlying data model, provided integration mechanisms, supporting infrastructure, and software architecture. *Together, these processes, and the aspects of computational models affecting them, comprise the domain construction framework.* The purpose of this chapter is to

describe the basic features of this framework. In Chapters 4 though 6, this framework will guide our analyses and comparison of different design environments.

This chapter begins by reviewing how high-level domain construction processes involve the use and change of design languages. Next, what I mean by "computational model" is discussed. The core of this chapter examines the processes involved in domain construction – use, elaboration, and modification – looking at what are their subprocesses and how these subprocesses depend on various aspects of computational models.

DESIGN LANGUAGE ENACTMENT

Our empirical analyses indicated that the central constituent of a domain is the shared understanding between domain practitioners that enables a community to use tools in a practiced way to create design products in a consistent manner. This shared understanding was called a "design language." Design languages consist of a vocabulary of domain distinctions, one or more representational systems highlighting important aspects of and relationships between distinctions, and evaluative knowledge and processes for judging or assessing the qualities of a design product. While creating individual design products (use), designers change the design language used to create these products (evolution) (see Figure 3.1).

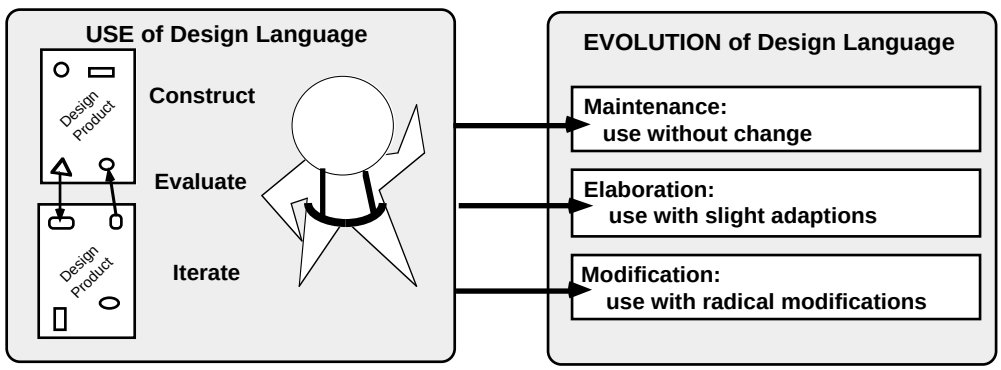

Figure 3.1: Relationship between use and evolution of design languages.

Use refers to the basic iterative processes of constructing design products, evaluating these products against various design goals, and feeding the results of evaluation back into the next construction cycle. The design language guides the use of tools to create design products in a consistent manner.

However, the world the designers operate in is not static; designers continually refine design products and thus evolve the underlying design language. Three basic patterns of evolutionary change to design languages were identified and characterized: maintenance, elaboration, and modification. Maintenance occurs when individuals enact design languages using the (sometimes implicit) rules of substance and form without alteration. Elaboration occurs when individuals consistently but slightly adapt design languages to reflect new conditions. Modification of design languages occurs when individuals depart significantly and persistently from existing rules, such as when prose documents are replaced by tabular representations.

Domain construction refers to the processes involved in these three modes of design language enactment. These are the processes that design environments need to support in order to simultaneously accommodate both use and change. In this thesis, maintenance is subsumed under the category of *supporting use*. Supporting change is divided into the two subcategories of *supporting elaboration* and *supporting modification*. Together, elaboration and modification define a particular form of design-in-use [56] specific to design environments.

DEFINITION OF A COMPUTATIONAL MODEL

The processes of use, elaboration, and modification are complexly related in that supporting one process sometimes comes at the expense of another. Thus, it will be difficult for any particular design environment to support all processes equally well. As we will see in the following chapters, the ability of environments to support these processes depends upon several aspects of the overall computational

model. In this thesis, a two-part, socio-technical definition of a computational model is adopted. On the technical side, a computational model consists of a particular configuration or architecture of tools, data, infrastructure, and integration services. On the social side, a computational model may require special practices, skills, and knowledge by those using it. Each of these aspects will be introduced below.

Tools are the various components required to construct design products. Thus, tools minimally support creating and editing representational systems.

Data , in the most general sense, refers to the way information concerning domain objects is represented in the computational model. We will consider two major types of data representations: domain models and underlying data models.

Issues surrounding *domain models* include the representation of presentation (i.e., look) and semantic information (i.e., content and behavior) and the ability of the domain model to be modified. In this dissertation, domain models representing mostly presentation information are referred to as "shallow" models; domain models representing additional semantic information are referred to as "deep" models.

The underlying *data model* refers to the formalism used by the domain model such as the abstract data type [37] or object-oriented formalism [6, 112, 113]. Issues to be considered include the flexibility of the underlying formalism with respect to: (1) representing domain information at different levels of formality and (2) support for modifying (i.e., adding and removing) domain information. Shipman defined formalization as "the process of identifying machine-processable aspects of information" (p.285) [107]. He classified information according to three basic levels of formality: informal, semi-formal, and formal. Informal information is not machine-processable; an example of informal information is a text document.

With semi-formal information, often the structure but not the content is machine-processable. An example of semi-formal information is design rationale in the gIBIS format [14]. With formal information, both the structure and the content is machine-processable and an example is a domain model represented in an object-oriented formalism.

Integration services refers to the interfaces, mechanisms, or programming languages available for integrating or linking domain information across design product and tool boundaries [24]. Issues to be considered include the granularity of integration supported; i.e., collections of objects, single objects, or parts of objects.

Infrastructure refers to the common parts of the computational model shared by tools or other services. Issues to be considered include how much of the domain model and integration services are provided by the infrastructure (and therefore shared by all the tools) versus provided by individual tools.

Architecture refers to the configuration or structural pattern of components (such as tools, data, and infrastructure) and connectors (such as integration links) [37, 105]. Issues to be considered include how robust the overall configuration is; i.e., can parts be significantly changed without adversely affecting other parts.

In addition to the technical aspects of a computational model, we will also consider the social or organizational aspects. For instance, often times, enabling practitioners to enrich or modify their computational environments requires *new skills* or *knowledge* on their part [36, 38, 80, 119]. As noted by Guindon, extending or programming software typically requires integrating knowledge from many different problem domains [49, 51]. Thus, when analyzing what knowledge or skills are required by a model, I will attempt to classify the required knowledge into categories such as application problem, programming language, tool architecture, etc. Examining the skills and knowledge required in this manner will better enable us to postulate

what *organizational support mechanisms* are required or would facilitate domain construction processes.

Thus, the computational model aspects to be considered in this thesis are:

- Which *processes* are supported or favored by the computational model?
- What *technical aspects* of the computational model help or hinder a particular process?
- What *new skills or knowledge* are required by the computational model?
- Is it likely that designers possess the necessary skills and knowledge or is *organizational support* required?

In the following sections, each of the domain construction processes of use, elaboration, and modification will be discussed from two perspectives. First, the detailed subprocesses involved in each of these categories will be derived from our empirical studies, other existing design studies, and theories of design. Second, the aspects of computational models that may affect these subprocesses are enumerated and questions and issues to be considered in the following chapters are posed.

SUPPORTING USE

In their daily work, designers *use* design languages to create design products. Use of design languages may be tacit; i.e., the rules of substances and form may be implicit and unstated. Or, use may be explicitly understood such as when a group shares standardized design representations. Use can even be formalized such as when tools support the creation of specialized, machine-interpretable design representations. Subprocesses of use include *constructing* individual design products, *evaluating* designs, and *iterating* the design (Table 3.1). Each of these subprocesses will be considered in turn.

Constructing

The basic design process followed by the communities in Chapter 2 was construct-evaluate-iterate. As we saw in Chapter 2, the design products constructed in these communities were based on a range of ill-defined to well-defined design languages. When design languages were ill-defined, the design products created were informal (not machine interpretable) design representations such as prose documents. As design languages became more well-defined, more explicit and even formal representations such as database tables began to emerge. And, as shown in both the multimedia and protocol analysis stories, design products often contain mixed informal and formal elements.

Table 3.1: Subprocesses and computational model considerations in supporting "use".

Supporting USE			
Domain Construction Processes	*Constructing* individual design products.	*Evaluating* designs. • inspecting • analyzing • relating • envisioning and experiencing	*Iterating* the design by continually changing design products. • maintaining consistency
Computational Model Considerations	• Can the tools or infrastructure be enriched with a domain model at the presentation level? • Does the data or domain model support the full formality spectrum? • Do the tools support mixed palettes for reusing design languages?	• Can the tools or infrastructure be enriched with a domain model at the semantic level? • Can tools or the infrastructure be enriched with awareness of relationships between design products; i.e., are there integration services?	• Does the provided integration services include notification mechanisms signaling when changes have been made?

To promote the use and maintenance (i.e., use without change) of design languages, individual designers should be supported in constructing representations using existing domain vocabularies as building blocks. For instance, the voice dialog designers constructed design products from vocabulary items such as voice menus, prompts, and messages into flow charts. Minimally, these vocabularies must be formalized at the presentation level and made available for reuse. For instance, a prompt vocabulary item from voice dialog design could be

defined as a rectangle with embedded text. Using domain vocabularies as building blocks has been theorized to have several benefits. First, it supports practitioners in conceptualizing their problem by giving them pertinent objects to interact with instead of low-level system-oriented primitives [28, 64, 99]. Second, it facilitates beneficial iterative design construction [41, 48] by not requiring practitioners to continually build up domain-specific units from low-level primitives [64]. However, the challenging issue is that sometimes all or parts of design languages are ill-defined. Thus, tools supporting constructing need a flexible data model capable of accommodating domain vocabularies at mixed levels of formality.

Evaluating

Designers do not simply construct representations, they also evaluate designs to see if design goals are being met by the artifact being constructed. Evaluation refers to the activity of analyzing a design to see if it conforms to various criteria and constraints [8]. For instance, the voice dialog designers continually evaluated their design representations for compliance with existing user interface guidelines. These designers used a combination of visual inspection and mental simulation to aid in their evaluation process. For experienced designers or small designs, these practices work fine. However, as designs grow large and complex, relying on visual inspection can be problematic as important features and relationships become difficult to spot. Relying on mental simulations can also be problematic. In her studies of software designers, Guindon found that designers had difficulty simulating designs due to cognitive limitations with mentally managing relationships between parts of the design [51]. Also not all designers are equally experienced; in the voice dialog community, many newly hired designers lacked detailed knowledge of the interface guidelines and thus did not know some of the relevant evaluation criteria.

Also, in all communities, a single design representation was insufficient to support their design process. As noted by Norman, a good representation emphasizes the important objects and

relationships and de-emphasizes the less important things [82]. However, there are many important aspects of any particular design and no single representation can show them all equally well. As a result, these communities created multiple representations that made different aspects of the design more inspectable and readily apparent. However, multiple representations created an evaluation burden; i.e., members in all communities reported that going back and forth across the representations was hard work, both manually and cognitively.

Thus, as noted by others, relying solely on experience and practices is fraught with potential error and designers could benefit from tools that support their evaluation activities [9, 10, 27, 29, 50, 78]. Tools supporting evaluation can help designers with analyzing design representations, relating design representations, and simulating the design. To support analyzing and simulating, design environments need to be able to parse and execute design products; this requires design products to be machine interpretable and based on a formal model of the domain. Supporting relating also requires that the relationships between design products be formally represented in the design environment.

Iterating

Following an iterative design process is not only desirable [40]; it is often necessary, because existing requirements change and new ones are uncovered as design proceeds [15, 48]. As indicated by the multimedia and voice dialog design stories, complex relationships exist between the various parts of different design representations. As designers continually reconstruct individual representations, inconsistencies with related representations are introduced. That is, changes in one representation trigger required changes in other representations, a phenomena Green called "viscosity" [42]. Viscosity hinders iterative design by imposing tedious maintenance and cognitive burdens on designers that result in design errors [42, 51]. Thus, in addition to the evaluation burden, multiple representations created a maintenance burden. Two design communities reported that this maintenance burden negatively affected their iterative design

practices. The problem was that the tools used by the designers did not support them at all in maintaining consistency across representations; the tools did not support locating affected objects or making the necessary changes.

Supporting iterative design requires design environments to contain a formal model of both domain objects and relationships between objects. Additionally, environments should provide a notification mechanism to signal design products (or the tools used to create them) that objects in them may have changed.

SUPPORTING ELABORATION

While use of design languages is a fundamental process, as we saw in our design stories, evolution of design languages was also important to the communities' overall practices. This section and the next will analyze the change processes of elaboration and modification observed in these communities.

Elaboration occurs when individuals consistently but slightly adapt design languages to reflect new conditions. These changes to design languages occur all along the explicitness spectrum as designers gradually enrich design products and tools with awareness of design languages. Subprocesses of elaboration include *articulating and refining domain vocabularies*, *establishing relationships* across design representations, and *extending* domain models and tools (Table 3.2).

Articulating Vocabularies

As discussed in chapter 2, design communities articulate domain vocabularies through the processes of *graphical refinement* and *naming*. Graphical refinement and naming help designers make the transition from tacit to explicit design languages as previously tacit assumptions become explicitly incorporated into design products.

Table 3.2: Subprocesses and computational model considerations in supporting "elaboration".

Supporting ELABORATION				
Domain Construction Processes	*Articulating* domain vocabularies; i.e., the tacit to explicit transition: • graphical refinement • naming (e.g., defining simple objects by specifying name / look mappings)	*Refining* domain vocabularies; i.e., the explicit to formal transition: • object refinement (e.g., defining complex objects containing multiple, named components by specifying whole / part mappings)	*Establishing relationships* within and across design products: • correspondence • equality • transformation • heuristic	*Extending* domain models and tools: • object behaviors • tool extensions
Computational Model Considerations	• How flexible is the data model? • Can vocabularies be graphically refined in situ in design products? • What special skills or knowledge are required?	• How flexible is the data model? • Can formalized objects continue to undergo refinement at the class and/or instance level? • What are the effects of refinements on existing design products? • What special skills or knowledge are required?	• What integration services are provided? • Are integration services provided by individual tools or the supporting infrastructure? • What granularity of integration is supported? • What special skills or knowledge are required?	• What extension mechanisms or languages are provided ? • Can behaviors and other procedural actions be associated with domain objects? • Can tools (menu, commands, etc.) be extended with new functionality? • What special skills or knowledge are required?

Graphical refinement refers to how designers incrementally elaborate on the look of domain distinctions within a particular design product. An example of graphical refinement is the way the protocol analysts used fonts and formatting commands to emphasize different aspects of their analysis document. Graphical refinement helps designers make the transition from tacit to explicit design languages as previously tacit assumptions become explicitly incorporated into design products. Supporting graphical refinement requires tools to have a rich and flexible set of formatting commands. Ideally, tools would enable designers to refine distinctions in situ; i.e., by directly operating on them in the design product using direct manipulation.

Naming enables designers to assign a meaningful identifier to a class of distinctions. At the articulation stage, naming establishes a mapping between an identifier and a particular look or presentation. The voice dialog designers created names for some of the distinctions in the initial textual specifications. These names were explicit in the sense that they embedded a key in the textual design product specifying the mapping between graphic looks (presentations) and distinction types. However, some tools supported designers to go a little further and enrich tools with awareness of these name / look mappings. These mappings are a form of *simple* object or data type definition. An example is how the protocol analysts used named styles in WORD to classify their vocabulary items and to associate a specific look with particular classes of items. Once defined, WORD made these distinctions available in a palette (pulldown menu) for later reuse. At the most minimal level, supporting naming requires tools to enable designers to enrich design products with keys specifying name / look mappings. Preferably though, tools can be enriched with these mappings. Essentially, mechanisms supporting a form of prototype instance definition [66] are required. Prototype instance definition is a type of object-oriented model where existing object instances are used as templates for creating desired classes. Such a mechanism would allow designers to incrementally elaborate on specific instances of vocabulary items from within a design product and when ready, save the elaborated instance as a class for later reuse. Ideally, creating such a class should not preclude further elaborations.

Refining Vocabularies

Design communities refine domain vocabularies by incrementally defining *complex* objects containing multiple, named components; i.e. by specifying or modifying whole / part mappings. Refining domain vocabularies in this way helps designers make the explicit to formal transition as design products and tools are enriched with more detailed models of design languages. The voice dialog community specified whole/part mappings at both the explicit and the formal levels. At the explicit level, voice menus in the later flow chart representations had been substantially refined to have more subparts since their

introduction in the initial flow chart representation. These designers went a step further and enriched their database tool, used for constructing the table representation, with formal (machine-processable) awareness of these whole / part relationships. We also observed that even after objects had been formalized, their evolution was not over. Designers continued to refine both the look (presentation) and content of formalized domain objects by adding new parts, modifying existing parts, and removing parts. As the voice menu example shows, evolution was not a simple, linear march towards progressively more elaborate objects. Many steps focused on simplifying existing representations by removing or refining existing parts; e.g., the "phrase" attribute in the phrase table representation (Figure 2.3B) did not appear in the initial flow chart menu representation (Figure 2.3C).

To support refining domain vocabularies, tools need to be able to be enriched with domain models containing whole/part mappings. This requires tools to have an underlying data or object model capable of representing such relationships such as a frame [125] or object/attribute [113] type of representation. The data model must be flexible enough to accommodate further refinements in the form of adding new parts and removing new parts. A critical issue is how changes to the data model are reflected in existing design products based on previous versions of the model; i.e., are these design products invalid or can changes to the data model be smoothly reflected in the products?

Establishing Relationships

As designers articulate and refine domain vocabularies, they also continually *establish relationships*, based on these evolving domain vocabularies, between different design products. This differs from maintaining relationships in that "establishing" focuses on how relationships come into being whereas maintenance just examines pre-existing relationships. In our design stories, several types of relationships were identified – correspondence, equality, derivation, and transformation. Correspondence relationships indicate which parts of different design products are related; no attempt is made to relate to

the content of the distinctions, only their place in the design product. The remainder of the relationships pertain to the content of domain distinctions. Equality relationships indicate which vocabulary items and their subparts should have identical content. Transformation relationships indicate that content in one representation can be mechanically derived from a related representation without recourse to domain-specific knowledge. Heuristic relationships require content in one representation to be transformed through the application of domain-specific knowledge to create or point to related content in another representation. These different forms of relationships reflect different positions in the explicitness spectrum: correspondence is primarily a tacit form of relationship whereas transformations are more formal relationships.

In our stories, designers had established relationships of all four types across their design products. However, it was their practices that maintained these relationships between design products; their tools provided no support. Ideally, integration services would be provided for establishing and maintaining these relationships. These services can be provided by individual tools or by the supporting infrastructure. All of these types of relationships require: establishing, notifying, and navigating facilities. Establishing refers to how the initial relationship between design products is established. Notifying refers to how the environment signals designers or tools that domain objects involved in relationships have changed. Navigating refers to how the environment supports moving from one end of the relationship (e.g., design product A) to the other end (e.g., design product B). A critical issue is the granularity of integration supported; i.e., can relationships be established between parts of domain objects (fine-grained) or can only entire objects (coarse-grained) be related? In our design stories, many relationships were very fine-grained, such as the complex relations between the flow chart and table representations in the voice dialog story.

Extending Domain Models and Tools

A fourth type of elaboration process is the *extending* of domain objects and tools. Extending is very formal form of elaboration probably involving programming. One form of extension is the ability to extend objects in domain models with procedural attachments. Procedural attachments can be used to give objects behaviors (e.g., make objects do something during design simulation). Another form of extension is modifications to design tools such as adding new functions or menu commands. Environments supporting such extensions need to provide an open, extensible architecture that includes a flexible data model and powerful programming facilities, perhaps in the form of general-purpose programming languages.

The elaboration processes described above involve issues that have been considered under the various rubrics of end-user modifiability [38], end-user programming [64, 80], and tailorability [70, 75]. Thus, it should be noted that many of the issues and techniques surrounding the subprocesses of establishing relationships and extending domain models and tools involve open research questions in HCI and AI. While the concepts are related, elaboration is not synonymous with either end-user modifiability or programming. Elaboration refers to the processes involved in *both* enriching design products *and* tools; whereas end-user modifiability and programming deal exclusively with enriching tools. The elaboration processes described here can be considered a subset of the general tailoring processes identified by Morch [75] – customization, integration, and extension – specialized towards enriching design environments with design languages.

Our design stories show that elaboration is a predominant domain construction process. However, if conditions significantly change in the domain, elaboration processes may be insufficient; incremental changes to existing design products may be inadequate for expressing newly emerging domain concepts. When existing representational systems significantly break down, modification processes come into play. Figure 3.2 summarizes the processes involved in elaboration and modification.

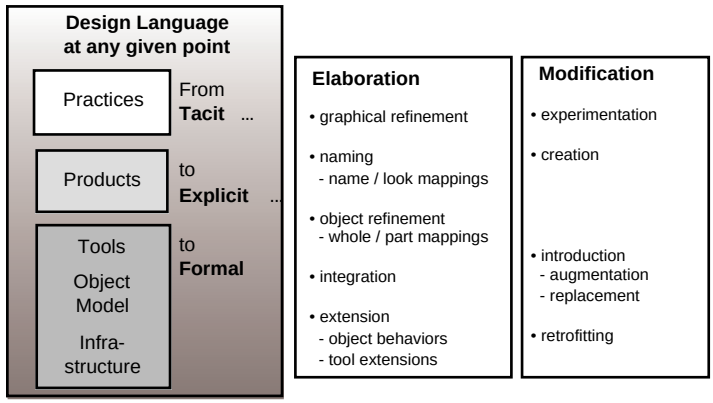

Figure 3.2: Processes supporting the evolution of design languages from tacit to explicit to formal.

SUPPORTING MODIFICATION

A hallmark of dynamic domains is representational instability. While elaboration processes are the most frequent manifestation of instability, modification is also an important factor in the overall domain construction process. Modification of design languages occurs when individuals depart significantly and persistently from existing representational systems. Subprocesses of modification include *experimenting* with and *creating* new representational systems, *introducing* new representational systems into the design environment, and *retrofitting* design products created with old languages to conform to new or significantly modified design languages (Table 3.3).

Table 3.3: Subprocesses and computational model considerations in supporting "modification".

Supporting MODIFICATION			
Domain Construction Processes	*Experimentation* with and *creation* of representational systems.	*Introduction* of new representational systems - either augmenting or replacing existing systems.	*Retrofitting* of products created with old languages to conform to new design languages.
Computational Model Considerations	• Are new tools required? • What knowledge or special skills are required?	• How difficult is it to integrate new tools with existing ones? • How much of the domain model is lost when an old tool is replaced?	• Can design products in old languages still be opened? • Does the model manage versions of design languages? • Is there support provided for merging versions?

Experimentation and Creation

In all three design stories, experimenting with and creating new representational systems played an important role in the evolution of the communities' design languages. The protocol analysts created a single column text representation to better support their analysis process. The lead multimedia designer created the hypermedia outline representation to support collaborative design of the title's content. One voice dialog designer created flow charts and tables because he suspected stakeholders were no longer reading the thick textual specifications. In all cases, experimentation with new representational systems was motivated by the realization that existing representations were inadequate for supporting some important aspect of the design process. In the protocol analysis case, the existing full page representation did not support the construction practice of annotating the protocols. In the voice dialog design case, the existing text representation did not adequately support communicating the design to other stakeholders. The multimedia case involved breakdowns in both construction and communication; it was tedious to modify the content of screens in Macromind Director and this hindered making real-time modifications to the content during collaborative design sessions with other stakeholders.

Design environments can support experimenting with and creating new representations in several ways. First, individual tools could provide a wide range of formatting and structuring features that enable them to support the construction of a range of representational systems. Many applications currently in the marketplace, such as word processors and spreadsheets, fall into this category. Second, if individual tools only support a narrow range of representational systems or an entirely new system is being explored, new tools may have to be located. Another consideration is the skills or knowledge required to experiment and create; i.e., can designers explore new representations at the tacit (presentation) level or must they formally define possible domain objects during exploration.

Introduction

Once a new representational system is created, the next challenge is introducing it into the design environment. As illustrated in our design stories, there are two categories of introduction – augmenting (in addition to) and replacing (instead of) existing representational systems. The protocol analysts replaced their full page text representation. The multimedia designers augmented their template representation with the hypermedia outline. The voice dialog designers did both. First they augmented the text representation with simple tables; when that design language significantly broke down, they replaced it with flow charts and complex tables.

There are many possible ways that design environments could support the introduction of new representational systems. When analyzing different computational models, I will focus on three aspects. First, if a new tool is required, is the architecture modular in the sense that it allows new tools to be "plugged-in" [54, 87, 121, 122]? Second, I will try to understanding how much of the existing domain model and integrating links are rendered invalid or have to be re-implemented when new tools or representational systems are introduced. Third, how difficult is it to integrate new tools or representations with those already present?

Retrofitting

Finally, when design languages have been significantly modified, design products created with old languages may have to be retrofitted to conform to the new design language. In some cases, retrofitting may be a necessary first step before re-use can occur; i.e. design products have to be brought up to date before any changes or additions can be made. Other times, retrofitting may be necessary to support evaluation and analysis activities. These activities often involve extensive comparisons across design products. These types of comparisons are much easier when the products being compared are based on the same representational system.

Retrofitting can be accomplished both manually and with environment support. Minimally, to at least enable manual retrofitting, design environments need to be able to open design products created with old design languages (i.e., presentation and semantic parts of domain models). To provide support for retrofitting activities, environment must first be able to store and manage previous versions of the domain model. Environments providing active support for merging versions need mechanisms for resolving differences at both the presentation and semantic levels.

LIMITATIONS OF THIS COMPARISON

Just because functionality is provided to enrich design environments with domain-specific concepts doesn't mean that designers will necessarily take this step. Empirical studies have shown that there are many barriers inhibiting practitioners from taking this step, including lack of time, lack of awareness of possibilities, and lack of motivation [13, 36, 38, 69]. Also there are the issues of cost, effort and benefit. Studies have shown that people will not extend their tools when the perceived effort outweighs the perceived benefit to themselves [46, 47]. Others have done more traditional cost/benefit analyses; comparing the actual monetary cost of buying or extending tools with productivity gains [63]. While these are all important considerations, they are not the focus of this comparative analysis. Instead, this analysis is focused on understanding the long term patterns of change in design languages and how these change processes can best be accommodated by computational design environments. As shown in previous chapters, understanding and supporting flexibility is an important step towards the success of design environments in workplace settings.

SUMMARY

In summary, Figure 3.3 presents the domain construction framework, relating ideal processes to computational model considerations. Ideally, tools *supporting use* would enable designers to

work continuously along this explicitness spectrum since, at any given time, different aspects of design languages exist at different points along a spectrum of explicitness: ranging from tacit understandings emerging in practices, to explicit domain distinctions reflected in products, to formalized domain models embedded in tools (Figure 3.2). Formal models are particularly required to support design evaluation and iteration. Thus, to support use, tools must allow domain distinctions to exist at different levels of explicitness. This requires environments to provide a flexible data or domain model.

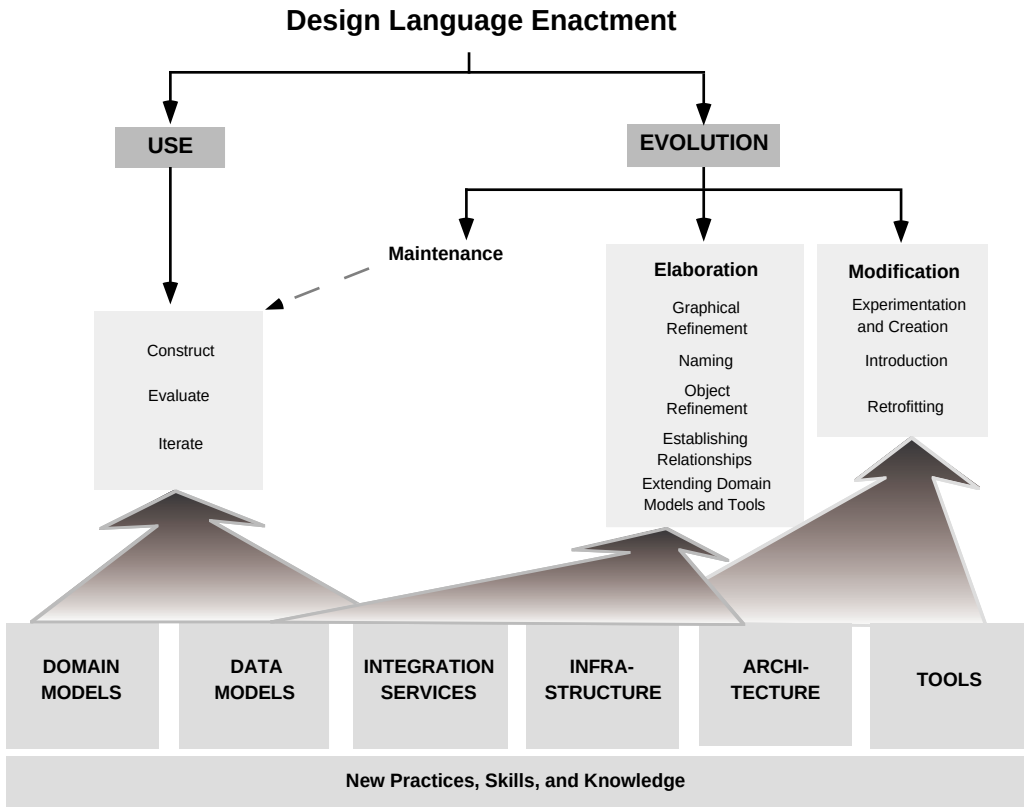

Figure 3.3: Overview of the Domain Construction Framework.

The framework relates a theoretical model of design language enactment processes with aspects of computational models that affect these processes. In the following chapters, the framework will be used to analyze the flexibility of two design environments based on different computational models, comparing their support for use and evolution.

Tools *supporting design language elaboration* should enable designers to incrementally change design products and tools by supporting a spectrum of tailoring functionality including: graphical refinement, naming, object refinement, integration across representations, and extensions such as adding behaviors. Some aspects of computational models affecting these mechanisms include: the type of underlying data or object model, the programmability of tools, and the integration mechanisms provided.

Tools *supporting design language modification* would ideally enable designers to experiment with new forms of representations, create new representations, introduce these representations into the design environment, and retrofit products created with old design languages to conform to new languages. Some aspects of computational models affecting these mechanisms include: the overall structure of the computational architecture, how much of the domain model resides in individual tools versus the infrastructure, and how much presentation information is tied to semantic information in the domain model.

In the following two chapters, the domain construction framework will be used to analyze and compare design environments based on two different computational models that seemingly occupying different ends of the support / flexibility spectrum. The purpose of these analyses is to look at which particular aspects of these models help or hinder their use and overall flexibility. Chapter 4 will examine three design environments based on the Toolbelt model. In this model, designers assemble and evolve collections of generic, off-the-shelf software packages such as word processors, graphics packages, and databases to create different design representations. Chapter 5 will examine the Voice Dialog Design Environment, which is based on the domain-oriented design environment model (DODE). In the DODE model, knowledge-based systems support the creation of specialized design representations. These two chapters will be similarly structured, beginning with a description of the general computational model followed by an discussion of the specific design environment(s) based

on the model. Next, project timelines from each design environment will be examined; these timelines combined with the resulting design environment will then be analyzed from the domain construction perspective.

CHAPTER 4

THREE TOOLBELT ENVIRONMENTS

In our three design stories, we observed an emerging software development context – the toolbelt model. In this context, many professionals assemble and evolve collections of generic, off-the-shelf software tools to create "systems." These systems may serve to support their ongoing work practices (e.g., the voice dialog design story) or they may be created for a specific project (e.g., the multimedia title story). We refer to these systems as "toolbelts" because each worker assembles her personal collection of tools just as a carpenter assembles a collection of hammers, screwdrivers, tape measures, etc. into a personal toolbelt.

Toolbelts are part of the end-user computing phenomena which is defined as the adoption, use, and development of software applications in support of organizational tasks by personnel outside of the organization's information systems department [11]. Currently, such end-user computing applications account for 50% of all corporate computing cycles in the United States; many researchers project end-user computing to account for most (75%) processor cycles by the end of the decade [11, 73]. Thus, the toolbelt model is part of the entrenched status quo and will probably continue to be so in the future.

While toolbelts are the entrenched model, they are also not without their faults. In this chapter, we'll analyze the toolbelt model's support for domain construction to better understand its strengths and weaknesses. This chapter begins by describing the general toolbelt model. The core of this chapter examines three specific toolbelts studied as part of this research. Finally, we generalize across

experiences with each of these toolbelts to analyze the general Toolbelt Model according to the domain construction framework presented in Chapter 3.

THE TOOLBELT MODEL

As illustrated by the design stories, a major part of a designer's job is to create and evolve external representations of the design being constructed [15, 103]. These representations occur in many forms such as textual, tabular, or diagrammatic. Often, these design professionals assemble collections of off-the-shelf software tools as needed to create the necessary design representations.

Typical off-the-shelf software tools include word processors, spreadsheets, databases, graphics packages, and flow charting tools. These applications are "generic" in that they are not targeted at supporting any particular domain. Instead, they are focused on supporting the creation of a particular type of representation. For instance, spreadsheets provide a lot of support for making tabular representations while flow charting packages make it relatively easy to construct node-link types of representations.

I refer to these software collections as "high-tech toolbelts" in analogy to the toolbelts used by craftspeople such as carpenters (see Figure 4.1). Carpenters assemble personal collections of hammers, screwdrivers, tape measures, etc. into toolbelts as needed to support a particular task. Toolbelt's are personal in that they are under the craftsperson's control (i.e., he or she chooses what tools are in it) and they go where the person does (i.e., when a carpenter changes sites, he takes his toolbelt with him). Often, a craftsperson will have several different toolbelts to support different tasks.

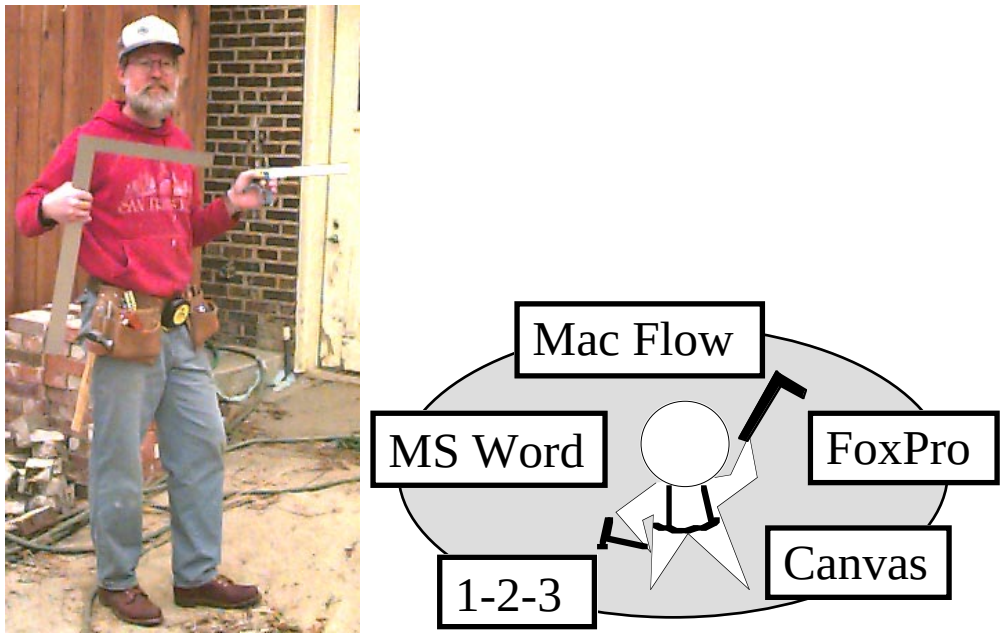

Figure 4.1: Toolbelts – Traditional and High-tech

Designers assembles collections of software tools to support different tasks (right side) just as carpenters assemble collections of screwdrivers, tape measures, etc. into toolbelts (left side).

For instance, Tom (Figure 4.1, left side) spends a lot of time working on Victorian homes. He has two toolbelts – a small one for touch-up work and a larger one for more complex tasks such as repairing wooden clapboard. When new tasks are encountered, he initially tries to apply old tools to the new task. For instance, he initially tried to use a simple hand held scraper to remove the paint off the clapboard. But due to the sheer volume of wood, he found he needed a more specialized tool that better supported this particular task. As Tom's understanding of the new task changes or new tools are discovered, he replaces old tools with new ones. Thus over time, the composition of tools in a toolbelt evolves to better support the tasks being performed.

Likewise, high-tech toolbelts (Figure 4.1, right side) are personal collections of software applications that an individual has assembled and evolved to support particular tasks. As the following toolbelt stories reveal, we see many of the same phenomena with high-tech

toolbelts as we see with the more traditional variety: the emphasis on assembling and evolving to tools to support a particular task, applying old tools to new tasks, replacing old tools with new ones, and the personal nature of these collections.

In the upcoming sections, we'll revisit the design stories to examine how the designers' toolbelts helped or hindered domain construction processes. For each design story, we'll look at three things. First, we'll look at how things ended up; i.e., the key design representations and the toolbelt used to create them at the end of the project or analysis period. Next, we'll see how things got to be that way by examining project timelines from a domain construction perspective. Afterwards, we'll critique the specific affordances and hindrances of the particular toolbelts used in each story. In the final section, we will try to generalize across each of these three stories and analyze in detail the toolbelt model's support for domain construction processes.

TOOLBELT 1: PROTOCOL ANALYSIS

As shown in Figure 4.2, the toolbelt used by the protocol analysts was very simple, consisting of only one word processing application – WORD. However, two different representations were being constructed with this tool. The primary representation was the single column protocol analysis document described in Chapter 2. This representation was a structured text document consisting of explicitly designated subsections and numerous domain objects. Towards the end of the project, another representation, "analysis views", was created. This tabular representation was an alternative view of a subset of the domain objects shown in the single column representation.

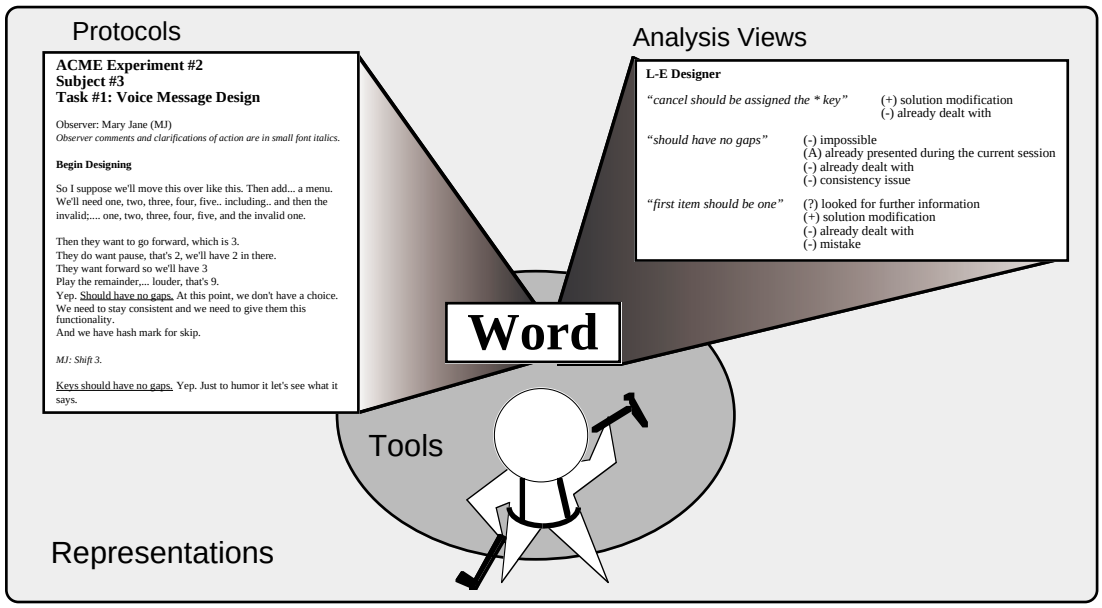

Figure 4.2: The toolbelt and representations used by the protocol analysts.

Examining the Timeline

The high-level sequence of domain construction processes occurring in the protocol analysis project are shown in Figure 4.3. In this story, the evolution of their design language was primarily driven by the changing needs of different project activities. However, affordances and limitations of WORD also impacted the evolutionary process.

The project consisted of three main activities. During the first activity (experiment #1), three protocols were collected and analyzed. WORD was chosen because it was the de facto standard word processing tool used by all group members. The first two protocols used the full page representation. However, as soon as analysis activities were attempted, the limitations of the full page representation were apparent. The single column representation was created and used for the remainder of that activity. The flexibility of WORD supported the significant restructuring of the overall form necessary to create the new representation. The first two protocols were then retrofitted to conform to the new design language.

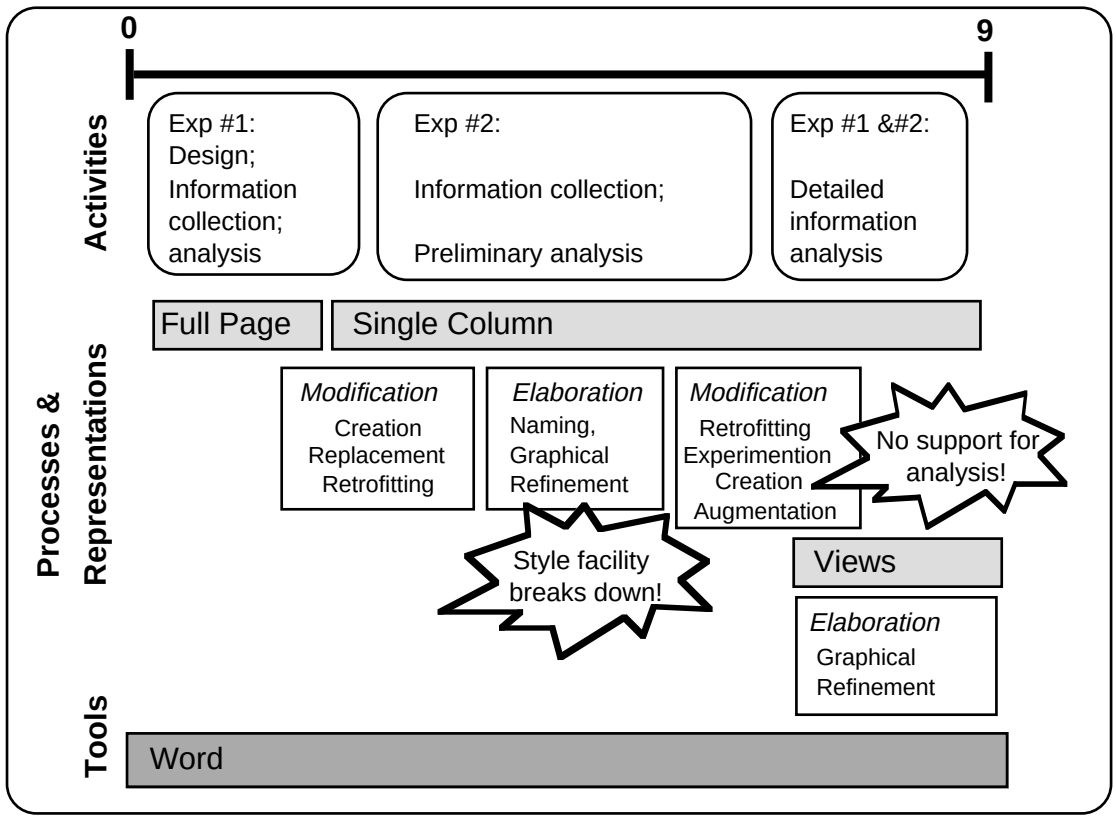

Figure 4.3: Protocol analysis project timeline.

The project lasted approximately 9 months, during which 9 protocols were collected and analyzed. The timeline on the top is provided for a sense of relative scale only and should not be used to make absolute temporal measurements. During this time, the analysts evolved a design language consisting of two representational systems and numerous vocabulary items. Only large scale changes in representational systems are shown here (light gray bars).

The second activity began about one month later (experiment #2) and six more protocols were collected. This activity started with the state of the single column representation at the end of activity 1. As each protocol was collected, more vocabulary items were explicitly added to the representation. In the beginning, WORD's style functionality was used to support this process. Styles are named classes of graphical looks (e.g., observer comments = 9 point, italics). Once a named style is created, it automatically appears in a pulldown menu at

the top of the document window; enabling this menu to act as a palette of reusable vocabulary items to choose from.

About midway through, mostly items pertinent to the general activity of protocol analysis had been defined and named (e.g., observer comments). As activities shifted towards some preliminary analysis, items specific to this round of experiments started to emerge. Attempts were made to use WORD's named style feature to define these items but it quickly broke down. Firstly, these items were small segments of text embedded in the middle of paragraphs; however, in this version of WORD, named styles only operated on entire paragraphs. A second problem was that some text passages participated in several different classes of vocabulary distinctions (e.g. acted as examples of both "anticipation" and "conflict awareness"). Since WORD's style sheet facility was designed to support document formatting, built into the facility was the constraint that text could not have two named formats at the same time. As a result of these early attempts, there are several extraneous named vocabulary items in the style sheet that were never used. Conventions were established to work around these limitations by using only graphical refinement (e.g., all underlined passages are examples of "critics"). Since these items were not explicitly named, they did not appear in the menu of reusable vocabulary items. Since the group relied on informal communication to relay what the mappings were, anyone outside of the initial project team looking at these documents will not know what all the symbols mean.

The focus of the third activity was to analyze the protocols from both experiments to look for patterns across subjects. However, the design language had evolved considerably by the ninth protocol and the previous eight protocols all had to be retrofitted to match the new design language before this could begin. The way WORD merges style sheets supported retrofitting fairly well. Old protocol documents could be opened and the new style sheet read in; the result was a style sheet combining both the old and new named items. New items overwrote old items with the same name, and objects in the document with this name were changed to reflect the new style definition. In this

retrofitting task, this type of overwriting was exactly the desired behavior. However, there were some other problems encountered while retrofitting. In the early documents, many items found in later documents were missing or were not explicitly named and only graphically refined. For instance, observer comments were depicted in 9 point italics for a couple of documents before a named style was created. Unfortunately, when retrofitting, WORD did not assist in reconciling unnamed formatted items with named items; i.e., suggest to convert all 9 point italic passages to be "observer comments." While it may have been possible to write macros to assist in this process, this step was never taken.

For the third activity, a new representation – analysis views – was required (see Figure 4.2). These views were a combination of existing information extracted from the protocol documents and new information. The representation was tabular, with items in the leftmost column corresponding to key events extracted from the protocol documents. Items in columns to the right, though new information, were related to information in the protocol documents, being classifications of subevents that occurred either before or after the key events. The extreme flexibility of WORD enabled us to create this new table-based representation without introducing a new tool into the toolbelt. However WORD did not help us to maintain the relationships between the two, closely related representations. And as the analysis proceeded, this lack of support for the interrelationships had two negative impacts. First, we had to rather tediously copy-and-paste or manually retype all the key events from several protocol documents to create the analysis view. However, the bigger negative impact was the cognitive and manual burdens incurred during analysis when classifying subevents. When reviewing classifications, there was no explicit link back to the corresponding subevent in the protocol document. You had to first find the approximate area in a typically lengthy protocol document and then browse until you found the proper subevent, which could be one of many. Likewise, we could not go the other way and select a subevent in the protocol document and be linked to the corresponding classification in the analysis view.

Analyzing the Toolbelt

Support for use. WORD provided moderate support for constructing individual representations. The palette of re-usable named, styles facilitated the consistent production of protocol documents. However, the workaround use of unnamed formats required the analyst to establish, remember and consistently apply existing conventions during document production. Where WORD really broke down was in supporting analysts at evaluating the protocols; i.e., their basic analysis task. This required extensive cross-design product comparisons for which the tool provided no support.

Support for elaboration. WORD's formatting features and style facility supported articulating domain vocabularies in many respects. The formatting features enabled domain objects to undergo continued graphical refinement. The style facility supported naming by allowing users to establish mappings between names and graphic looks. Styles can be defined by selecting an existing object in the text and using it as a prototype for a style definition or a style can be constructed from the style sheet by choosing from available formatting commands. Once defined, the style is made available for reuse in a pulldown menu in the top level toolbar.

However, there were some drawbacks to WORD's support for elaboration processes. At the time, WORD did not support character styles; i.e., only entire paragraphs could be formatted with named styles. This severely limited the articulation of domain vocabularies. While, a workaround was available and used (i.e., unnamed formats), the mappings between names and looks were never explicitly recorded which could hinder future analyses of the documents. Where WORD particularly fell short was in supporting establishing correspondence and equality relationships between the different design products. What was really needed was a simple mechanism allowing links to be established between related parts of the two representations. To be effective, the links would have to support bidirectional navigation. Given the quantity of potential links (about 75) and their dynamic

nature (the analysts were continually debating and changing the classifications), the overhead to create these links would have to be low.

Support for modification. WORD's flexibility in terms of formatting and editing commands supported experimentation with and creation of new representations without requiring new tools. The way styles could be merged across two documents supported important retrofitting activities when old design products were changed to reflect new design languages.

TOOLBELT 2: MULTIMEDIA TITLE DESIGN

In contrast to the previous example, the toolbelt used by the multimedia title designers was quite complex. Figure 4.4 shows the toolbelt used by the lead designer. It consisted of eight tools. Two tools – WORD and Director – were used to create the major design representations. Director was used to create two representations used for designing the look or form of the title – the template and the score representations. WORD was used to create the hypermedia outline representation for designing the content of the title. The remaining six tools were used for special effects such as textured backgrounds and for creating information in alternate modalities such as audio or video. The outline and template representations were used to design how all these various effects and information modalities would fit together into a coherent and entertaining whole.

In this story, some of the individual nature of toolbelts starts to become apparent. Contrary to the previous story, not all project participants shared identical toolbelts – some participants focused on a subset of the tools shown, other participants used these tools and others that are not shown. Thus, there was only a partial overlap between the toolbelts of all participants. To some extent, this reflected the division of labor within the project and peoples' individual tool preferences.

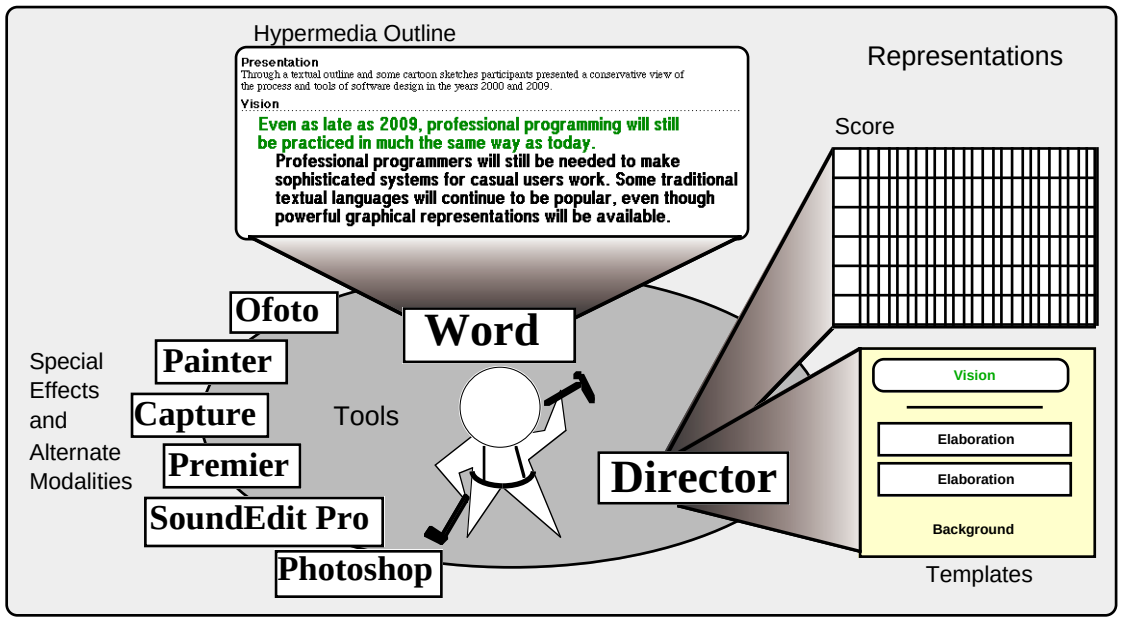

Figure 4.4: The toolbelt and representations used by the lead multimedia designer.

Examining the Timeline

Figure 4.5 shows the sequence of domain construction processes occurring in the multimedia title project. Similar to the protocol analysts, the evolution of their design language was driven by the changing needs of different project activities. However, in this story, tool affordances and hindrances also affected evolution.

In the beginning, designing the form of the title was the main activity. Early on, they decided to use Director to implement the final title and also to design the initial form. However Director did not support the range of visual and auditory effects wanted, so much initial effort was spent experimenting with different tools and sequences of tools to get the desired effects. Particularly troublesome was an effect needed for the opening screen and other screen transitions in the title. The designers wanted a series of five Polaroid-like images to slide onto the background from different trajectories and end up in non-rectangular orientations. This proved very difficult to do since Director only supported animating rectangular "sprites". A large part of the initial project effort was spent experimenting with workarounds in Director and sequences of tools to support this one effect.

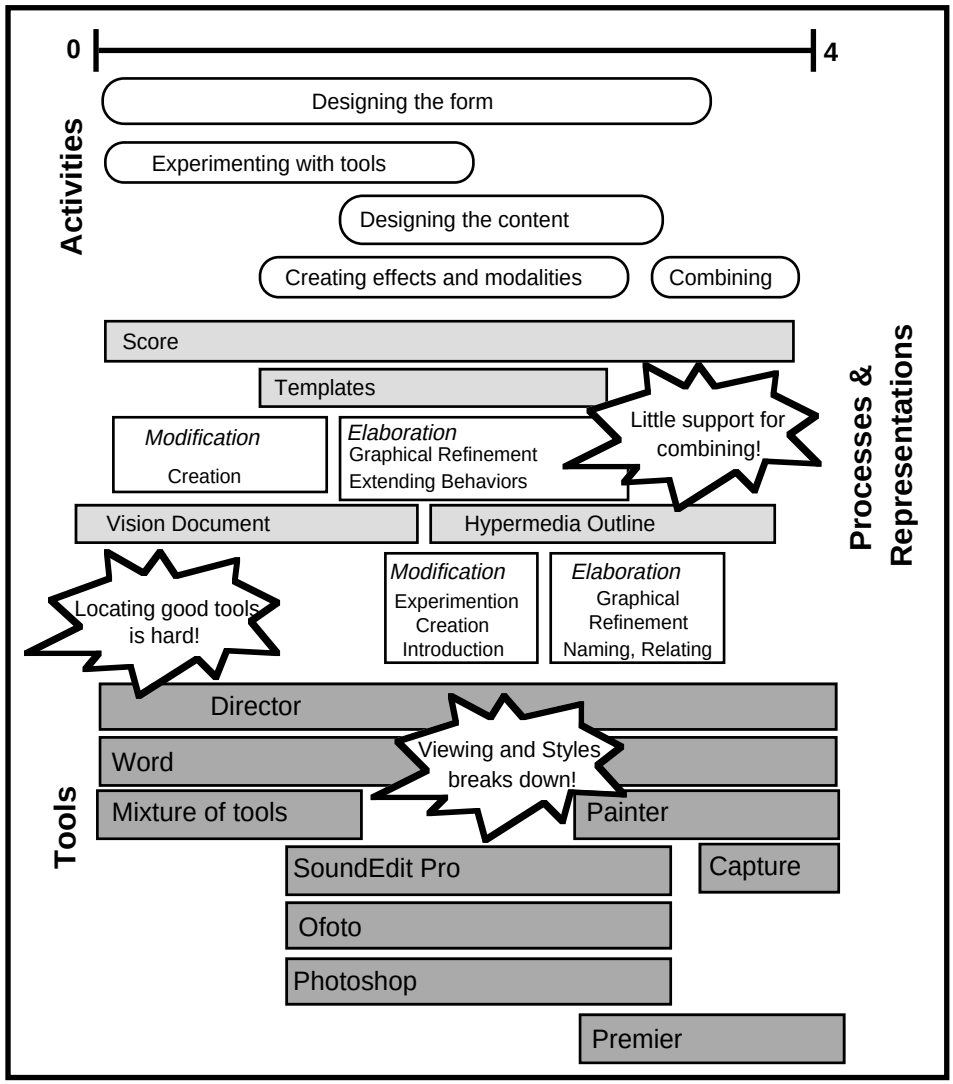

Figure 4.5: Multimedia title project timeline.

The project lasted approximately 4 months, during which time a multimedia title was created to be shown at an upcoming conference [18]. The timeline on the top is provided for a sense of relative scale only and should not be used to make absolute temporal measurements.

Once the various techniques for creating the desired effects and modalities had been ironed out, they could be applied *en masse*. The middle part of the project was concerned with producing all the various effects and information modalities to be included in the final title. This involved things like scanning in and touching up images,

recording audio information, and preparing video snippets. A file structure was created to contain all these information "pieces" in an orderly fashion.

Some of elaborating on the form of the title involved programming Director. However, the central part of the title's final form was 23 screens showing textual entities called *visions* and *elaborations*. Six graphic template representations were created when designing this part of the form. These templates specified background colors, auditory information, and the position, font, and color of the different kinds of textual information (i.e., the content). During collaborative design sessions, this text needed to be continually modified. However, modifying text in Director was slow and cumbersome, so the lead designer created a hypermedia outline representation to support collaborative design of the title's content. During the time it took to create and co-evolve the template and hypermedia representations, the team articulated seven domain vocabulary items, one of which was never used.

WORD, which had previously been used to collect and manage vision statements from workshop participants (vision document representation), was now used to create the new representation. The lead designer wanted to make the content representation closely resemble the template representation in order to support envisioning and to minimize the cognitive load when performing modifications across the two representations. Using WORD's formatting features, the font, color, and size of the text in the hypermedia outline was made to match the final look in the template. In this story, the designers did not make use of WORD's style facility to articulate domain vocabularies; instead they simply embedded the vocabulary names in their design product. One reason for doing this is that the outline facility only operated on standard styles such as heading 1, heading 2, etc. The designers wanted to use the outline facility for viewing parts of the content representation, similarly to how it would appear in the graphic templates form. Thus, the designers had to apply the standard style names to their domain items and remember that heading 3 was used

for *vision* statements and heading 4 was used for *elaboration* statements, etc. A side effect of using standard headers was that all domain objects with the same heading name (which determined their position in the outline view) had identical formats. As a result, completely different objects ended up looking alike (e.g., "summaries" looked like "visions").

During the final phase of the project, all the various parts of the title had to be integrated into the score representation in Director. These parts included the audio recordings, video clips, scanned images (all organized in the file structure), and the text segments (found in the hypermedia outline representation). Several difficulties were experienced during this "combining" phase. First, each media element had to be individually imported into Director, using the Import File command, which yielded a "cast member." Each resulting cast member was then placed into the score and the member's numeric reference had to be manually adjusted. These numeric references are used when executing a title to determine which cast members to load at any given time. Unfortunately, these references were absolute, which meant that if a new cast member needed to be inserted in the middle of the score, all subsequent numbers had to be manually adjusted. Thus, it was very tedious to make such alterations.

More difficulties arose when transforming the text in the hypermedia outline into the graphics templates. One problem was that only flat ASCII text could be transferred between WORD and Director; i.e. all formatting information such as font, size, and color was lost. The designer compensated for this by including this information as part of the specification for each of the six templates. However, the main problem was that the sequence of operations required to transform the data was lengthy and this needed to be done 23 times; once for each screen. This complexity arose because the designers wanted screens to have a textured background and such texturing was not supported by Director. For each of the 23 screens, the following transformation process was manually followed:

(1) Paste vision statement from WORD into provided area in Director template.

(2) Paste elaboration statements from WORD into provided areas in template.

(3) Take a screen snapshot of the filled in template using Capture.

(4) Import the snapshot into Painter.

(5) Change to the textured background using Painter's "Apply surface texture" command.

(6) Copy the result and paste it back into Director.

While it only took a few days to do this for all the screens, the lead designer reported that they were reluctant to make changes to any text, i.e., iterate, once this process was complete. They would only make changes for "glaring errors such as typos."

Analyzing the Toolbelt

Support for use . Their particular toolbelt broke down during the final "combining" activity in that their individual tools provided them with no support for constructing (combining their separate products into the final score) and iterating the design. There were two interrelated problems: lack of support for named domain vocabularies in Director, and a lack of support for relationships across design products. Directors provision of only absolute numeric references, instead of relative naming, made it difficult to modify the score. This, combined with a lack of integration services in the various tools, hindered the designers ability to iterate their design.

Support for elaboration. The designers' tools enabled them to quickly articulate domain vocabularies, mostly at the presentation level, in the template and the hypermedia outline representations. The lead designer reported that this took at matter of days. They were able to easily co-evolve the look of the text segments in the outline

representation to visibly match the same distinctions in the template representation using WORD's formatting features.

However, they experienced a minor breakdown when trying to use WORD's outline feature to create an alternate view of the hypermedia representation; it only operates on standard style names. Thus to be able to manipulate their domain objects in this view, the designers were required to name their domain objects the standard style names.

The tools particularly broke down in their lack of support for establishing relationships across products. Ideally, a programmatic transformation process could be described between WORD and Director such that any changes to the text in WORD would be automatically flushed through to Director.

Some off-the-shelf applications enable such links to be established using direct manipulation. Because there are 23 screens, it is unlikely that direct manipulation would be a satisfactory approach. A few applications provide high-level event scripting languages such as Apple Script. In theory, such a scripting language would have been able to capture and automate the transformation process between WORD and Director. In practice however, most of the tools being used did not support such scripting.

Other options for establishing relationships between applications are integration protocols and integration languages such as Visual Basic, DDE (Dynamic Data Exchange) or OLE (Object Linking and Embedding) [74] on Windows-based machines or OpenDoc [87] on Macintosh platforms. However, two problems arise with these integration approaches. First, similar to high-level event scripting, many existing applications do not support the same protocols in compatible ways. Second, these integration services are low-level programming languages targeted at developers, not end users. To create the transformation script required by the multimedia designers would require a considerable amount of programming and knowledge from several different areas including object-oriented programming,

imperative programming, and detailed internal knowledge of the applications being integrated. One sample OLE script taking data from a database and linking it into a table in a spreadsheet required about nine pages of code specifying low level event details such as opening and closing applications, setting window focus, and redrawing windows [110]. Given the one-off nature of this project, it would not be cost effective, in terms of time and effort, to go to such lengths to tailor tools to better support work practices. In this case, "customizing themselves" to act as the glue binding their individual tools into a coherent system was the only viable alternative.

Support for modification. As in the previous story, we saw that designers were able to create new representations specialized towards their specific needs. We also saw how these designers rapidly evolved their toolbelt to better fit their project-specific needs. In this story, multiple tools were required to create all the parts (special effects and modalities) associated with the final score representation. These designers spent much time figuring out which particular tools they needed in the initial part of the project. As this experience shows, while the toolbelt model offers significant freedom, it is not without a price. End users sometimes need to expend significant effort and energy in searching for, learning, and experimenting with various tools. The lead designer said he was surprised at how long it took to figure out how to achieve the non-rectangular animation effect and that he probably "spent too much time" on it.

TOOLBELT 3: VOICE DIALOG DESIGN

The toolbelt used by some of the voice dialog designers is shown in Figure 4.6. It consists of three tools – WORD, FoxPro, and TopDown. TopDown, a flow charting application, is used to create the primary flow chart representation depicting the content and structure of the phone-based interface being designed. This representation is used for communicating the design to other stakeholders such as marketing and customer support. WORD is used for creating structured document representations such as high-level design specifications and detailed

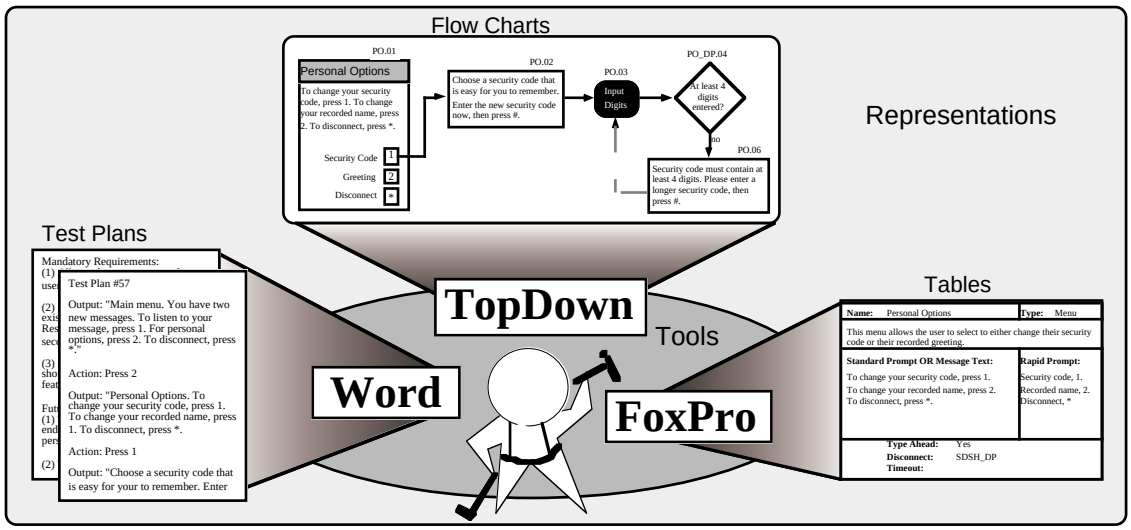

Figure 4.6: The toolbelt and representations used by some of the voice dialog designers.

test plans. The specifications are constructed for marketing; the test plans are constructed to verify the final application against the design. FoxPro, a database application, is used for creating a structured table representation depicting low level details of the interface. This representation is primarily constructed for the vendor company that implements the final application.

Examining the Timeline

The voice dialog design story is both similar and different to the previous two stories. In the previous stories, small groups of people came together for an individual project and disbanded when the project was completed. All toolbelts and design languages created pertained only to those single project. The voice dialog story chronicles the experiences of part of a design group who received a series of voice dialog design contracts over a several year period. Some of these design contracts were for new applications; others were for redesigns to parts of existing applications. An additional difference was the dynamic group composition, old designers left, new designers joined, and new marketing and vendor stakeholders also entered the picture. Thus, the story covers many design projects carried out by different people.

While the various members of the group would agree that they were all doing voice dialog design, the specifics of what it meant to design in this domain continually changed as design languages, products, tools, and practices co-evolved. The top of Figure 4.7 shows the high-level sequence of domain construction processes occurring in the voice dialog design story. The bottom half of Figure 4.7 shows the accompanying evolution of tools and representational systems. What is particularly different in this story is how representational systems differed across designers as they tuned them to their specific needs and how these representational needs drove designers' toolbelts to evolve in different ways.

Prior to 1991, primarily textual design specifications were being used. Designer A was the lead designer and had been working with the same marketing and vendor organizations on a series of additions and redesigns to an existing product for quite some time. Using WORD, she had evolved a structured text representation that graphically distinguished three, named domain objects and the name/look mappings were specified in an explicit key incorporated into her design product. She augmented this representation with simple flow charts and simple phrase tables created in MacDraw.

In early 1991, Designer B was hired to design one of the most complex applications to date. In his mind, the textual representations did not scale up to the new task. He pioneered two new design representations – detailed, complex flow charts and tables – to replace the existing text representation and he assembled an initial toolbelt to support making these representations. He was able to use an existing tool, MacDraw, to experiment with the new flow chart representation. Essentially, this use of an old tool enabled him to preview the new representation without any monetary cost and little application learning cost. He did have to locate a new tool, FoxPro, to create the new complex table representation as the layout was too complicated for Excel at that time. WORD was still used for the remaining textual parts of the design specification.

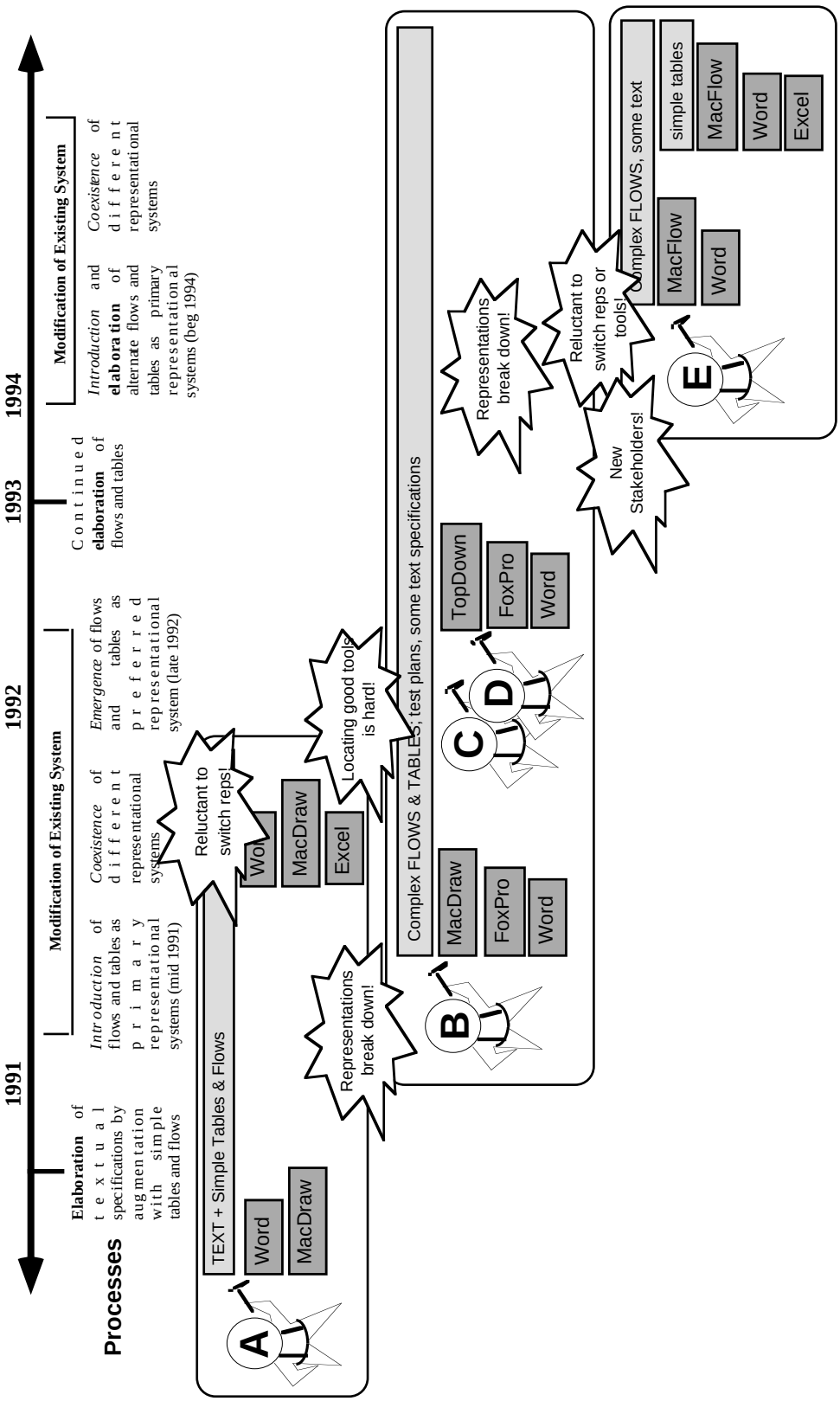

Figure 4.7: Voice dialog timeline showing five designers engaged in a series of projects over a three year period.

Designer A was reluctant to adopt these new representations for several reasons, First, she was reluctant to switch representations on the other stakeholders which had grown accustomed to receiving and interpreting her text representations. Second, constructing flow charts in MacDraw was very tedious and time-consuming work. Designer B reported looking at flow charting packages and finding that they did not have the capacity to accommodate large designs. Designer A continued to do things her way and even introduced a new tool into her toolbelt, Excel, that better supported making the simple phrase tables.

The marketing group decided to cancel the project Designer B was working on and the new representational systems were not used until the next year. In the middle of 1992, two new designers were hired into the group to work on a lengthy series of additions and redesigns to the same product that A had previously been working on. About the same time, Designer A transferred to a new project. The new designers, C and D, together with Designer B, sat down and discussed the pros and cons of B's new representational systems. In the meantime, a new tool had been found that better supported making the flow charts – TopDown. Designers C and D decided to adopt B's representations and his, now modified, toolbelt.

C and D began to elaborate on B's representations. In the flow charts, both designers used the new, more elaborate voice menu representation (though D did not adopt shaded titles) and both incorporated the new inputter object. However, they both, individually, began to experiment with new graphic distinctions which the other one did not adopt. One problem with the flow chart representation was that all the links clutter it up and make the flows difficult to read. Both designers took advantage of different features in TopDown to create different graphic conventions that they thought made the links in the flow chart easier to read.

The table representation was reorganized to improve the layout of information and new fields were introduced. Both designers adopted the new table layout. However, Designer D began to elaborate on the table representation by introducing a few new domain object types and by modifying the pseudocode notation for expressing conditionals. Designer C was not too concerned with these innovations because she structured her practices to avoid making the table representation. She considered it not particularly helpful for designing the application interface and avoided constructing the tables until the very end; in fact, she usually found a student intern to construct this representation for her. Designer D experienced his own problems with the table representation. He completed all the tables (a little more than 100!) associated with a design when the marketing group decided the interface needed to be substantially changed. The proposed changes required modifying virtually all of the tables; a very tedious and laborious manual process. At that time, the feeling was starting to emerge that it was best to delay constructing the tables until you were through iterating the design.

Designers C and D continued to work on this related series of designs, though C eventually moved to part time status in the project. In the beginning of 1994, Designer E was hired to create a series of small applications for the same marketing group (but different people in the group). He was also working with a different vendor organization. Before E was hired, a contractor had been doing some of the initial design work and was using a somewhat different flow chart representation than that used by Designers B, C, and D.

Designer E decided not to adopt the representations used by the others for several reasons. First, similar to Designer A, he also used the argument that other stakeholders were already used to the alternate flow chart representation. However, he had an additional reason. The new vendor organization used a different development platform and this platform had a flow chart-like interface. His different flow chart representation was formatted to, somewhat, resemble the vendor's flow chart interface. He was thinking about promoting this similarity

of look even more to make it easier for the vendor. Initially, Designer E did not adopt any form of the table representation because he felt its complexity was overkill for his simpler applications and his partner in the vendor organization did not need it.

Designer E also decided not to adopt the other designers' toolbelt. He chose instead to use MacFlow since he used this tool in his previous job appointment. By mid 1994, Designer E decided he did need a table representation of some sort (though still a different one) and he was experimenting with using Excel. Shortly thereafter, he reported that he might have to eventually move to from Excel to a database package for more flexibility.

Analyzing the Toolbelt

Support for use. Overall, designers were able to construct the necessary design representations. In a few cases though, they needed to adjust their practices to compensate for their tools' lack of support for articulating domain vocabularies. One problem was the lack of support for naming and reuse in the flow chart tools. While it is not especially difficult to create a prompt unit from scratch (merely a rectangle and some text), it is a considerable effort to create the elaborate voice menu representation from scratch every time. One designer had evolved a copy/paste strategy to deal with this and whenever a new menu was needed, opened an existing design or scrolled until a menu could be found.

A bigger problem stemmed from the lack of support for relationships across design products which hindered iterative design. Similar to the multimedia story, these designers had to act as the human glue manually binding their representations together. Besides being hard, due to the complexity of the interrelationships, it was also error prone. Designers had to remember which nodes they changed in the flow chart and figure out which tables would be affected by this change, and then carry out the changes without making mistakes. This manual burden hindered iterative design by raising the cost and effort associated with iterating the design. Contrary to the multimedia story,

these designers did not have the luxury of deciding not to iterate. Instead, we saw that these designers began to modify their practices and defer constructing the table representation until later in the process. One possible negative consequence of this deferment is that some design information, found only in the table representation, is not considered until (possibly too) late in the design process.

Another problem, not discussed in the previous story, was the "medium gap" between the visual design products and the final audio interface. Designers had some difficulties evaluating designs because it was difficult for designers to look at a visual design product and imagine what it would sound like. They needed to actually hear the audio prompts in sequence. The solution adopted was to have simulation builders in the group construct prototypes of parts of the design using specialized simulation programming languages. While this approach had several benefits, one problem was the ensuing delay (days to weeks) between creating a design product and experiencing the prototype. However, this approach was really the only option given the level that existing tools could be extended.

Support for elaboration. Some tools such as WORD and FoxPro supported designers to articulate domain vocabularies and to evolve the structure of their representational systems. Other tools such as TopDown and MacFlow supported designers to graphically refine their flow chart representations but did not support them in taking the next step to enrich their tools with some awareness of the domain objects at either the presentation (name/look) or semantic (whole/part) levels. Specifically, these tools did not support naming or any form of object refinement.

As discussed above and in Chapter 2, complex relationships existed between the flow chart and the table representations used by Designers B, C, and D (see Figures 2.7). At that time, neither TopDown or FoxPro provided any facilities establishing relationships across design products. Later versions of FoxPro now support the OLE object linking protocol. However, similar to the multimedia case, neither existing

direct manipulation or programmatic interfaces are appropriate. With over 100 tables in a design and about a dozen relationships per table, establishing each relationship using direct manipulation is not viable. Also, some of the relationships are transformation ones that cannot be specified via direct manipulation techniques. One could argue that, if they located a flow charting tool that supported the same object protocol, the programming effort required to link the two representations using OLE might be worth it since this is a reoccurring need. This is debatable since, in mid-1994, both the community using the table representation and the popularity of the representation itself seemed to be waning. However, a bigger problem was that when asked in 1994, none of the designers had ever heard of OLE or its competitor protocol, OpenDoc. It is difficult to take advantage of features that you don't realize exist.

Support for modification. Similar to previous stories, we observed how designers assembled the toolbelts necessary to support the creation of their design representations. In this story, we particularly observed how designers took advantage of the flexibility of the model to create representations specialized towards the needs of their specific stakeholder partners. However, this level of variance in individual toolbelts was not without its costs.

First, these individuals bore the burden of locating, learning, and experimenting with the necessary tools. One benefit of the toolbelt model was that designers could use old tools in new ways to preview new representational systems without incurring much monetary or learning cost. However, as we saw with the flow charts of Designer B and the tables of Designer E, this was often a temporary measure until a new tool could be found. Unfortunately, designers were not always successful in locating the necessary tools and sometimes other designers delayed adapting new representations due to lack of tool support.

Second, the group undoubtedly suffered some overhead associated with managing such as diverse work environment such as difficulties

sharing current designs and difficulties reusing past designs. When Designer C began to redesign a feature in an existing product originally designed by someone in the same group, she had to reverse engineer the design as the first step. She spent considerable time retrofitting; i.e., putting the existing interface into a flow chart representation readable by TopDown.

REFLECTIONS ON THE GENERAL TOOLBELT MODEL

This section generalizes across the stories to consider how the general Toolbelt model supports domain construction. First, we'll examine support for processes involved in use, elaboration, and modification. Next, we'll discuss the impact of technical aspects of the computational model on these processes. Finally, the new practices, skills and knowledge required by the model are considered.

Support for USE

Table 4.1 summarizes toolbelt support for use processes of constructing, evaluating, and iterating designs. The toolbelts we observed supported the construction of design products but did not support evaluation or iteration. This lack stemmed from most tools' inability to be enriched with "deep" domain models. In general, individual tools only supported very shallow domain models. None of the graphics packages allowed any form of domain model at all. Shallow domain models have a limited representation of domain vocabularies, focusing mainly on presentation (name/look mappings) and limited semantic information. Deep models provide flexible whole/part mappings, relationships between objects, and procedural attachments to objects; deep models are required to support evaluation and iteration activities. Representations of relationships between objects are required to support iterating and evaluating across design products. Procedural attachments are required to provide simulation and other active behaviors supporting design evaluation. In summary, the toolbelt model mainly supports design languages at the tacit and explicit levels only; i.e. most of the design language resides in practices and design products only.

Table 4.1: Toolbelt support for USE.

Support for USE	Toolbelt Model
• full DL spectrum supported?	mainly tacit / explicit
• constructing	Yes
• relating	No
• evaluating - inspecting - analyzing - relating - envisioning & experiencing	Limited - visual inspection only
• iterating	No

Support for ELABORATION

Table 4.2 summarizes toolbelt support for elaboration processes. In general, the individual tools mainly supported designers to make the tacit to explicit transition by graphically refining design products. Some tools supported making the transition to formal domain models in limited ways, primarily through the definition of simple objects; i.e., the creation of name/look mappings. The only tool supporting the definition of complex objects with multiple, named components was the database tool. This tool enabled designers to specify whole/part mappings. However, due to its relational nature, all objects were required to have the same whole/part mapping. This limitation made it difficult to tell incomplete objects (individual database tables) from objects where all the various parts did not apply. None of the tools supported extending objects with behaviors or other procedural attachments. A particular shortcoming was the lack of support for any form of elaboration within graphics packages which are important tools for creating many design products.

None of the design communities took advantage of the integration services provided to establish relationships across design products and tools. There are several possible reasons for this. First, the integration

services provided are very uneven and ad hoc. Many tools, especially graphics packages, provide no services at all. Second, integration services are mostly left at the discretion of individual tools and not provided in the infrastructure (Figure 4.8). The result is that different tools supply different integration methods and interfaces, each of which must be learned. Third, to establish the type of relationships and the fine-grained granularity of relationships that we observed requires extensive low-level programming effort, which for one-off projects or dynamic domains may not be worth the effort.

Table 4.2: Toolbelt support for ELABORATION.

Support for ELABORATION	**Toolbelt Model**
• DL transitions supported?	mainly tacit to explicit
• graphical refinement	Yes
• naming - defining simple objects by specifying name / look mappings	Limited - No support in graphics tools
• object refinement - defining complex objects by specifying whole / part mappings	Very Limited - In few tools only
• establishing relationships - correspondence - equality - transformation - heuristic	Possible, but difficult - Limited Direct Manipulation - Collection of ad hoc mechanisms - Low-level programming
• extension - object behaviors - tool extensions	Limited - tool extensions

Support for MODIFICATION

Table 4.3 summarizes toolbelt support for modification processes. In general, the toolbelt model did very well at supporting modification processes. The flexibility of individual tools enabled designers to experiment with a range of representations without requiring new tools. The tools' support for graphical refinement enabled experimentation to occur at the tacit and explicit levels without forcing designers to enrich tools with formal knowledge of a design language which might never be adopted. Sometimes, designers used old tools in new ways to preview new design languages; this supported experimentation without the cost of learning or exploring new tools. However, other times new tools were required. As we saw in the stories, locating new tools required new tool shopping and exploring skills from designers.

Since most of the individual tools in the Toolbelts have no or shallow domain models, there is no appreciable side effects associated with introducing them. However, once an individual tool has been enriched with a significant domain model or effort has been spent to establish programmatic relationships across tools, then there is considerable costs associated with introducing tools. If an old tool is replaced out right, all the domain knowledge embedded in that tool is lost (Figure 4.8). If a new tool is added to augment existing tools, considerable effort must be spent up front to enrich the individual tool since there is no knowledge-sharing across tools in the Toolbelt model.

For the most part, the tools in the toolbelt model neither helped nor hindered retrofitting activities. Since most tools had limited awareness of design languages, retrofitting mainly involved re-opening old design products and reformatting them to graphically conform to new languages. Some tools with presentation-level domain models provided a limited form of design language versioning by storing these shallow models with each individual design product and providing mechanisms for merging different models.

Table 4.3: Toolbelt support for MODIFICATION.

Support for MODIFICATION	Toolbelt Model
• experimentation and creation	Yes - tacit experimentation - previewing - requires tool shopping and exploration skills
• introduction - augmentation - replacement	Yes and No - Yes, if shallow model - No, if deep model
• retrofitting	Limited. - Mostly manual. - Active support limited

Technical Aspects of the Computational Model

Figure 4.8 illustrates the generalized toolbelt computational model. where individual tools are used in isolation to construct particular design products. Tools are the centerpieces of this model; there is little infrastructure support beyond basic operating system services such as file management. Data models reside in individual tools; there is no provision for data sharing or integration across tool boundaries. For the most part, individual tools do not support evolving their general data models beyond simple presentation and semantic enriching.

Feiler and Wallnau refer to this type of model as a "coalition environment" because the integration services provided reflect political and economic coalitions between different vendors [24]. Coalition environments are characterized by the "point-to-point integration of coalition tool services with each other, rather than with underlying framework services" (i.e., infrastructure) pg.4 [24]. As a result, integration services tend to be ad hoc and control-oriented (as opposed to data-oriented) since this requires less coordination and consensus between coalition participants.

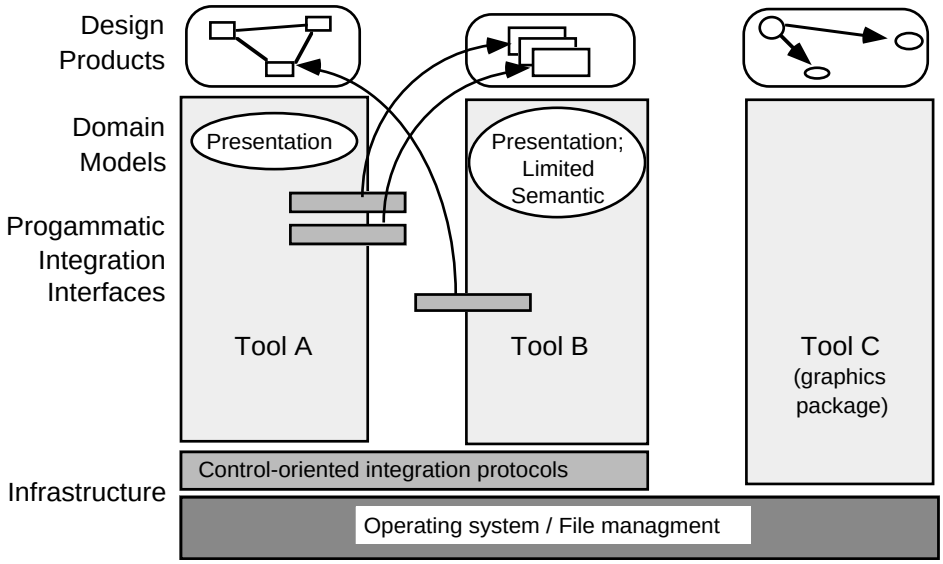

Figure 4.8: Generalized architecture of the Toolbelt computational model

In the personal computer arena, coalitions are coalescing around two different compound document architectures – OLE and OpenDoc. Compound document architectures provide inter-application communication protocols that specify a standard way for applications to expose their internal objects for extensibility and programmatic control [54, 74, 87, 121, 122]. OLE (by Microsoft, [74]) provides such facilities on MS Windows-based personal computers and is now beginning to appear in some Macintosh applications. OpenDoc (by the Component Integration Laboratory, [87]) is currently being implemented for the Macintosh and other hardware platforms. While these protocols provide some form of consistent across-tool infrastructure integration services, the way applications choose to support these protocols and the interfaces applications provide in this area are still highly individual. For instance, in one well-known "integrated suite" of four applications provided by the same vendor, each application provided a different level of OLE integration support and provided a different variant of programming language to access these integration services [110].

Overall, this type of architecture is very flexible in certain conditions. However, there is limited support for customizing,

integrating, and extending individual tools; where support is provided it is difficult since every tool offers different interfaces and extension languages. Also, since there is no data sharing, there is a lot of redundant effort since each tool must be enriched with its own version of the domain model. Once tools have been significantly enriched, either at the domain model level or the relationship (integration) level, then the model is no longer very flexible as all that elaboration effort is lost or invalid when the tool is replaced.

New Practices, Skills and Knowledge Required

The toolbelt model both enables new ways of working and requires new skills and new types of computational literacy. Table 4.4 lists the new practices required by the model and the new skills and knowledge needed to be successful.

Table 4.4: New Practices, skills and knowledge required by the Toolbelt Model.

	New Practices	**New Skills & Knowledge**
Minimal	• Locating Tools / Shopping • Experimenting with Tools • Assessing Tools	• expertise on tool availability • expertise on market trends • tool learning and exploration skills • tool evaluation skills (features, programmability, interoperability)
Preferred	• Customizing Tools • Extending Tools • Linking Tools	• detailed application knowledge • macro and scripting languages • application linking protocols

The toolbelt model creates a tool-mastery burden different from past models of computer-supported work. In the past, the tool mastery burden has focused on the difficulties users have learning to use a tool. While learning is still an issue, as discussed above, selection of tools and "gluing" of individual tools into systems are adding significantly to the burden on designers.

Minimally designers need skills associated with locating and assessing tools in order to assemble their initial toolbelts. Sometimes,

the search for new tools that supported specific objectives was unsuccessful. Designers do not know if the tools really do not exist or they simply were not able to find them. One problem is that keeping abreast of the software market is a full time job in itself. One large supplier of off-the-shelf software carries 2,000 items in stock and is prepared to special order from a selection of 10,000 items total. Two warehouse suppliers specializing solely in Macintosh products carry 2,000 and 5,000 items in stock respectively (hardware and software).

Also, designers need to experiment with and assess the suitability of tools for supporting specific work practices. This requires learning and exploration skills in order to evaluate the current features of a specific tool. However, many people lack well-formed strategies for these two activities [100]. Besides assessing just the features and interface, deeper assessments regarding the tools programmability and interoperability may also need to be made to estimate long term suitability. In addition, knowledge of market trends is also required to assess the longer term suitability of a tool. Questions such as will the tool manufacturer go out of business soon and will they continue to support this product need to be investigated. For instance, the voice dialog designers are slowly phasing out the use of one simulation tool because the manufacturer has gone out of business.

These skills and others associated with extending and linking tools are additionally required to support evolving toolbelts to better support work practices. As previously mentioned, currently the designers' practices bind the applications together into a design environment. The advent of commercially available compound document architectures, such as OLE and OpenDoc, indicates that the technology for automating some of this tedious and error-prone dependency management may soon be in place. However, simply providing the technology is insufficient for ensuring that it is properly understood and used. For instance, most designers probably do not know that architectures such as OLE or OpenDoc even exist or what their potential uses are. In the voice dialog group, none of the designers had ever heard of either architecture.

Also, a fairly detailed knowledge of such architectures is required when selecting tools to support the long-term evolution of work practices. For instance, how an application "supports" a protocol is quite a tricky issue in that there are many different ways that an application can provide support. In the case of OLE, many applications currently only output OLE objects and cannot accept objects as input. Other issues that need to be investigated prior to selecting a tool include granularity (e.g., Are objects entire flow charts or can individual nodes be an object?) and support for programmability (e.g., Can dependencies only be established by direct manipulation or can they be established automatically such as generating the corresponding table entry whenever a new node in the flow chart is created?) Answering these questions already requires a large amount of technical savvy, and these are just the questions to be asked when *buying* software tools. Actually *extending* the tools to interoperate using provided programming interfaces still remains to be addressed!

As the stories show, the designers themselves were able to adapt to meet many of the new demands of the toolbelt model. However, tuning toolbelts to better support work practices, beyond the limited forms we saw in our stories, would undoubtedly require organization support for the reasons discussed above. This issue of organizational support will be reconsidered in Chapters 6 and 7.

SUMMARY

This chapter illustrates how a large part of these designers' job is "designing their design process" which consists of their design languages, products, tools, and practices. Overall, the toolbelt model, where designers assemble and evolve collections of off-the-shelf generic software packages supported this co-evolutionary process. The toolbelt model supported domain construction areas such as modification and tacit to explicit elaboration processes. Support for some use processes, such as evaluation and iteration, were lacking. The toolbelt model may be monetarily cheap but it is not without costs. It

requires domain professionals to spend time locating and extending tools and it requires them to acquire new practices, skills and knowledge. The findings in this chapter will be used in Chapter 6, when comparing toolbelts with the DODE model. Chapter 7 will use these analyses on the strengths and weaknesses of the toolbelt model to inform the next generation of tool design.

CHAPTER 5

THE VOICE DIALOG DESIGN ENVIRONMENT

Innovative design support environments have been developed by several researchers [22, 25, 94, 111]. One thing these efforts all have in common is their emphasis on providing domain-oriented support. Domain-oriented systems embody a domain model of the entities to be manipulated and can use the model to provide active support to the designer. An example is the ProNet computer network design environment enabling designers to construct logical network maps showing the positions of workstations, printers, and gateways [115]. The system's domain model represents these common domain objects (i.e., workstations and printers), the relationships between these objects, and rules guiding device connectivity. The system uses the domain model to analyze logical maps for compliance with the connectivity rules.

As illustrated with the network design example, providing active computational support for individual design activities often requires enriching the system with an understanding of the domain; e.g. building a domain model into the system. However, building a rich domain model into a system might limit the system's flexibility; i.e., it might no longer be capable of dealing with significant, perhaps even small, changes in the domain. Thus, as discussed in Chapter 1, our intuition is that systems with rich domain models provide better design support, but offer less flexibility. The purpose of this chapter is to investigate these intuitions; particularly, to deepen our understanding of the particular aspects of a domain-oriented system that help or hinder its overall flexibility.

To investigate these intuitions, we'll examine in detail our experiences creating and using a particular domain-oriented system – the Voice Dialog Design Environment (VDDE). VDDE is based on the domain-oriented design environment (DODE) model, where specialized design representations and domain knowledge-bases are provided to support designers in constructing and evaluating design solutions. VDDE was constructed in collaboration with the same voice dialog design community discussed in Chapters 2 and 4. Its development was concurrent with the voice dialog toolbelt described in Chapter 4.

This chapter begins with a brief overview of the general domain-oriented design environment model and its theoretical underpinnings. The central core of the chapter details the experiences from the Voice Dialog Design Environment project. Next, these specific experiences are analyzed from the domain construction perspective. Finally, the chapter concludes by reflecting on how generalizable the VDDE experiences are to the general DODE computational model.

THE DOMAIN-ORIENTED DESIGN ENVIRONMENT MODEL

The general DODE model has been thoroughly described in numerous publications [25, 26, 28-30, 33, 34]. A brief overview of the general model is provided here. DODEs provide tools for creating design representations, information repositories for storing domain knowledge, and knowledge-based mechanisms that link the design representations and the stored domain information. DODEs instantiate the multifaceted architecture shown in Figure 5.1.

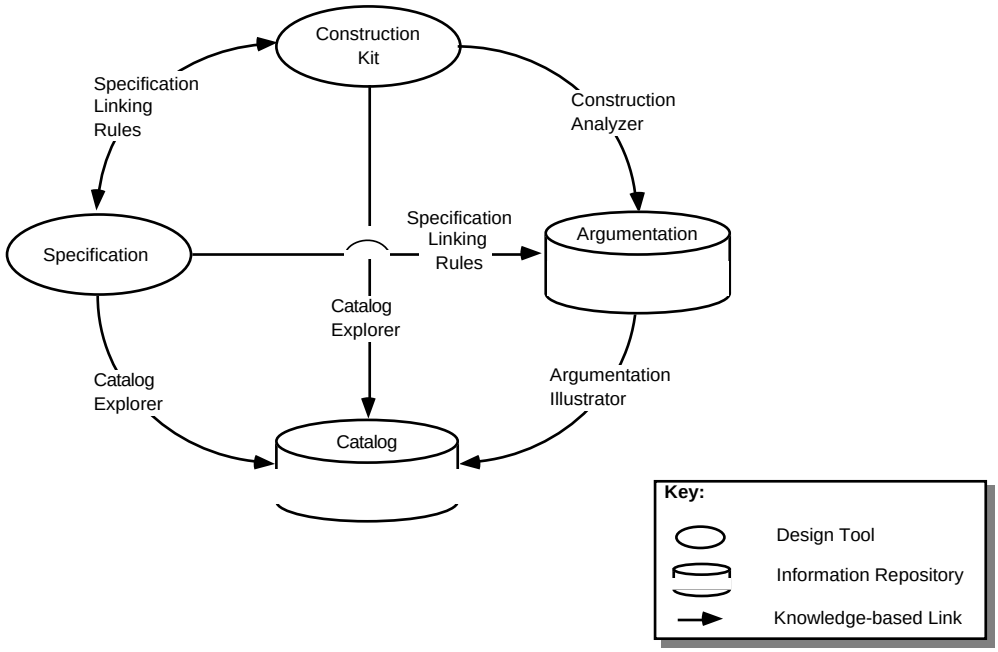

Figure 5.1: The DODE multifaceted architecture.

The multi-faceted architecture provides tools for design construction, information repositories, and knowledge-based links integrating these various components. Figure reprinted courtesy of J. Ostwald.

The architecture contains design creation tools in the form of a construction component and a specification component. The construction component is the principal medium for modeling a design. The goal of the construction component is to support design by composition and design by modification. To do so, it uses a construction kit approach [28], providing a palette of domain-oriented design units which can be arranged in a work area using direct manipulation.

The specification component allows designers to describe abstract characteristics of the design they have in mind. The specification provides the system with an explicit representation of the user's goals which can be used to tailor the activities of the knowledge-based linking mechanisms towards the specific design being constructed [76].

Design information repositories are provided in the form of argumentation, catalog, and simulation knowledge-bases. The argumentative hypermedia component contains design rationale [14, 72] that users can annotate and add to as it emerges during the design process. The catalog component [77] provides a collection of previously constructed designs and is intended to support reuse [89] and case-based reasoning [61, 62]. A simulation component (not shown) enables designers to carry out "what-if" exercises and simulate usage scenarios of the artifact being designed [98, 115].

The architecture is multifaceted because these components provide multiple representations of both the current design and underlying domain knowledge. The knowledge-based linking mechanisms serve to integrate these facets in the DODE architecture. Computer-based critics are the primary form of linking mechanism. Computer-based critics are made up of sets of rules or procedures for evaluating different aspects of a design; sometimes each individual rule or procedure is referred to as a critic [29, 33, 78]. The critiquing mechanism analyzes the design construction for compliance with the currently enabled set of critic rules. When a potential problem is detected, the critic signals the designer and provides entry into the argumentative hypermedia component where an explanation is located. It is left up to the designer to choose whether or not to modify the design in response to a critic message.

In the following sections, these DODE components will be considered in the specific context of the Voice Dialog Design Environment.

INITIAL TASK ANALYSIS: CHALLENGES FACING DESIGNERS

Voice dialog designers create software applications with phone-based user interfaces. Typical applications are voice information systems and voice messaging systems. These applications consist of a series of voice-prompted menus requesting the user to perform certain actions; e.g., "to listen to your messages, press 1." The caller issues

commands by pressing touch-tone buttons on the telephone keypad and the system responds with appropriate voice phrases and prompts. Designing in this domain means specifying the interface for an application at a detailed level. The designs are then usually given to some other person or organization to be implemented on specialized hardware platforms.

In early 1991, we began a collaborative research project with two voice dialog design groups within US West. One group specialized in the design of large-scale voice dialog applications such as voice messaging systems. This is the same design community discussed in Chapters 2 and 4. The other group, composed of voice dialog account executives, specialized in small-scale customized voice information systems. Some members of these groups were dissatisfied with their current design tools; i.e., the toolbelts discussed in the previous chapter. Two representatives for these groups approached us with a compelling presentation of the problems and challenges faced by voice dialog designers. This section reviews some of the findings from our initial task analysis that led us to believe that voice dialog design was a good candidate domain for the DODE approach.

In order to better understand the domain and the design challenges, members of the voice dialog design groups were interviewed, design sessions were observed, a few sessions were videotaped and analyzed, and existing tools and design representations were analyzed. Four core challenges facing designers were identified [9, 10, 98, 117]:

- Inconsistent Design Representations
- The Audio/Visual "Medium" Gap
- Knowing and Applying the Design Guidelines
- Coping with Conflicting Design Guidelines

Inconsistent Design Representations. In 1991, voice dialog applications were a relatively new design domain. Historically, applications had been small in scale, with most applications offering only a handful of features; e.g. providing three options to hear a

selection of recorded information. However, in the early 90's, voice dialog applications mushroomed in size. It is not unusual to have voice mail systems with 50-page instruction manuals, hundreds of features, and a two-year development cycle. Industry deregulation combined with advances in hardware triggered a rapid spread of voice dialog technology into new application areas. Key challenges facing designers, both then and now, are large increases in complexity and rapid innovation within the application domain. As a result of this rapid change, there are no standard design representations used by most voice dialog designers. One common representation is the traditional written specification. Some designers, in an attempt to deal with increasing complexity, have moved to graphical representations similar to flow charts. Both voice dialog groups participating in this project were using graphical representations. However, while these representations were similar on the surface, there were many differences both graphically and semantically.

The Audio/Visual "Medium" Gap. The design process of voice dialog applications is complicated by the "gap" between the visual medium of design representations and the audio medium of the end product. Halstead-Nussloch [53] identifies many important differences between screen-based (visual) and phone-based (audio) interfaces. Screen-based interfaces display information on the screen for a duration that is under user control. In phone-based interfaces, information is presented auditorally as spoken messages and signals and must therefore be processed serially with no memory aids whatsoever. Short term memory limitations are an important design consideration. It is difficult for the designer working with a visually-oriented, long duration design representation to mentally bridge this medium gap and envision the auditory, serial, short-term-memory constrained end product. Simply put, it is difficult for designers and end-users of these applications to anticipate what the final audio design will sound like by simply looking at a static, visual diagram. Thus, simulations are built that allow designers and end-users to directly experience the final audio interface. Unfortunately, using current software packages, a simulation for a simple design takes a professional

programmer several days to build; a simulation for a complex design can take a couple of weeks to produce.

Knowing and Applying the Design Guidelines. Building applications that conform to existing design guidelines requires knowing and being able to apply the guidelines. This was problematic for the group designing small-scale voice information systems. Their "designers" were not trained user interface designers but instead were small business systems salespeople. These salespeople went to the clients' place of business and co-designed a system on the spot using flow chart sketches and paper forms. For the most part, these people did not know the user interface guidelines and ended up designing systems that did not comply. Existing members of the group designing large-scale voice messaging systems did not have many problems in this area; primarily because they were the authors and maintainers of the guidelines document. However, the group's manager noted that about 50% of all newly hired designers had little previous experience designing phone-based interfaces. Thus, the first thing many new designers needed to do was learn the existing design guidelines.

Coping with Conflicting Design Guidelines. One challenge facing voice dialog designers is that their design task is influenced by many conflicting design objectives such as compliance with different sets of user interface guidelines (regional, national, and international) and the desire to create designs that are consistent with related products or existing applications in the installed product base. Unfortunately, many applications in the installed base predate the interface guidelines and so do not conform to these guidelines. Thus, the designer must make difficult trade-off decisions between these competing objectives.

In spring of 1991, we began to design and prototype a voice dialog design environment. The group creating small-scale voice information systems was disbanded before the end of 1991. However, the collaboration with the large-scale group continued until the middle of 1994. During this period, a voice dialog design environment instantiating many parts of the DODE model (except the catalog) was

designed, developed, and evaluated. From the beginning, the project's emphasis was on creating a combined construction / simulation component, targeted at providing consistent design representations and bridging the audio/visual medium gap. Later on in the project, a critiquing system was added containing encoded design rules, a hypermedia design rationale component, and a simple specification component. This addition was targeted at supporting designers to learn and apply the different user interface guidelines and to assist designers in making difficult trade-off decisions between the different guidelines and product consistency.

The remaining sections will describe the final VDDE system and our experiences creating the system. Analogous to the toolbelt stories in the previous chapter, I'll first describe the state of the system at the end of the project. Then, I'll discuss how it got to be that way by examining the project timeline. Finally, the resulting system and our experiences creating it will be analyzed from the domain construction perspective.

THE FINAL VDDE SYSTEM

The Voice Dialog Design Environment provides a construction kit that allows designers to quickly sketch out the flow of an audio interface by arranging domain-oriented design units such as voice menus and prompts into a flow chart-style representation. Designers can hear what their audio interface design sounds like by attaching audio recordings to components in the interface and simulating the design. Computational design critics embedded in the system watch designers' actions and comment on potentially problematic aspects of the design under construction.

The VDDE system, as it stood at the end of the project, is shown in Figure 5.2. The top two windows show the combined construction / simulation subsystem. The lower window and rightmost window show the critiquing subsystem. Each of these subsystems and the final system architecture will be described below.

The Construction / Simulation Subsystem

The construction / simulation subsystem is the core of the Voice Dialog Design Environment and is built using the Agentsheets visual programming substrate [94, 95, 99]. This section assumes the reader has familiarity with the Agentsheets substrate. This subsystem was targeted at solving the inconsistent design representations and audio/visual medium gap challenges. The DODE model, combined with the construction kit and visual programming capabilities of the Agentsheets substrate, suggested possible solutions.

To support the creation of standardized design representations, VDDE provides a construction component consisting of a gallery containing domain-oriented design units and worksheets (see Figure 5.2). Designers create designs by selecting voice dialog interface units from the gallery and arranging them in worksheets according to three standardized placement rules.

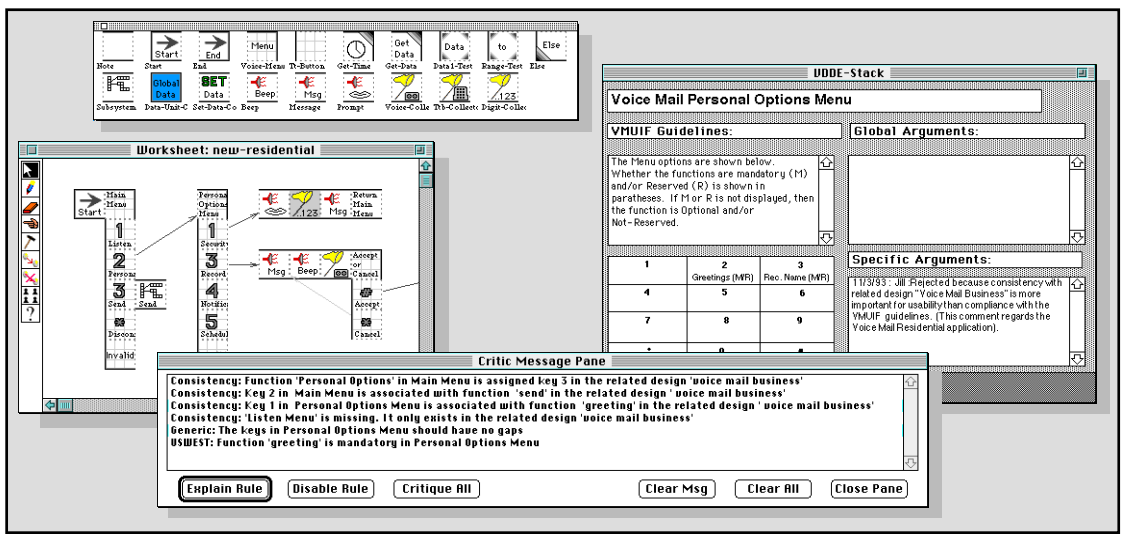

Figure 5.2: The Voice Dialog Design Environment.

Designers arrange design units from the gallery (top window) in a worksheet (left window) to create a graphic representation of the audio interface. A critiquing component analyzes the design for compliance with interface guidelines and product consistency. Problems are signaled in the critic message pane (lower window). The designer can elect to see the rationale behind the rule and can also add more arguments into the hypermedia knowledge-base (right window).

To support bridging the audio/visual medium gap, VDDE enables designers to construct executable design representations. Designers can attach sounds to design units in the worksheets and simulate the design at any time. Simulation consists of a visual trace of the execution path combined with an audio presentation of all prompts and messages encountered.

The gallery (Figure 5.3) consists of 19 voice dialog design units. Thirteen design units (top 5 rows) are based on the domain vocabularies found in the flow chart and table representations. These design units are more specialized than related units found in the flow chart representation; i.e., instead of one inputter unit, there are three design units for different types of input. Likewise, there are two units for creating menus, and five design units for expressing conditionals. This degree of specialization was necessary for the design representation to be executable. Towards the end of the VDDE project, the "Note" design unit was introduced mimicking generic flow-charting node capabilities. This design unit basically does nothing. Designers can replace the blank depiction with a small, three word label; when simulating the design, flow passes right through the unit.

Five new design units (bottom two rows) were introduced specifically to support design simulation. The Start, End, and Subsystem units direct the flow of the simulation process. The Subsystem unit was introduced to encourage designers to hierarchically decompose designs into modular subsystems; each unit represents a nested worksheet. When simulation control enters a Subsystem design unit, the corresponding nested worksheet is opened and control passes to that worksheets' Start unit. When the simulation reaches either blank space or an End unit, control returns to the calling worksheet. Two units – Set Data and Global Data – were added to manage data during design simulation. The Set Data unit is required for conditionals operating on data items, such as the Get Data unit, to work correctly during simulation. The Global Data unit is for debugging purposes; it enables designers to observe when data values change during the course of design simulation.

As in all other Agentsheets applications, each design unit is an agent having a specific look (bitmap depiction in the gallery) and a behavior (associated object class). In VDDE, many design units also have special methods and interfaces for configuring their various attributes. Most configuration methods and interfaces deal with recording and managing audio prompts. Figure 5.4 shows the typical parts of a VDDE design unit. Some design units, such as the prompt unit, allow the designer to replace or augment the graphic bitmap depiction with a small text label.

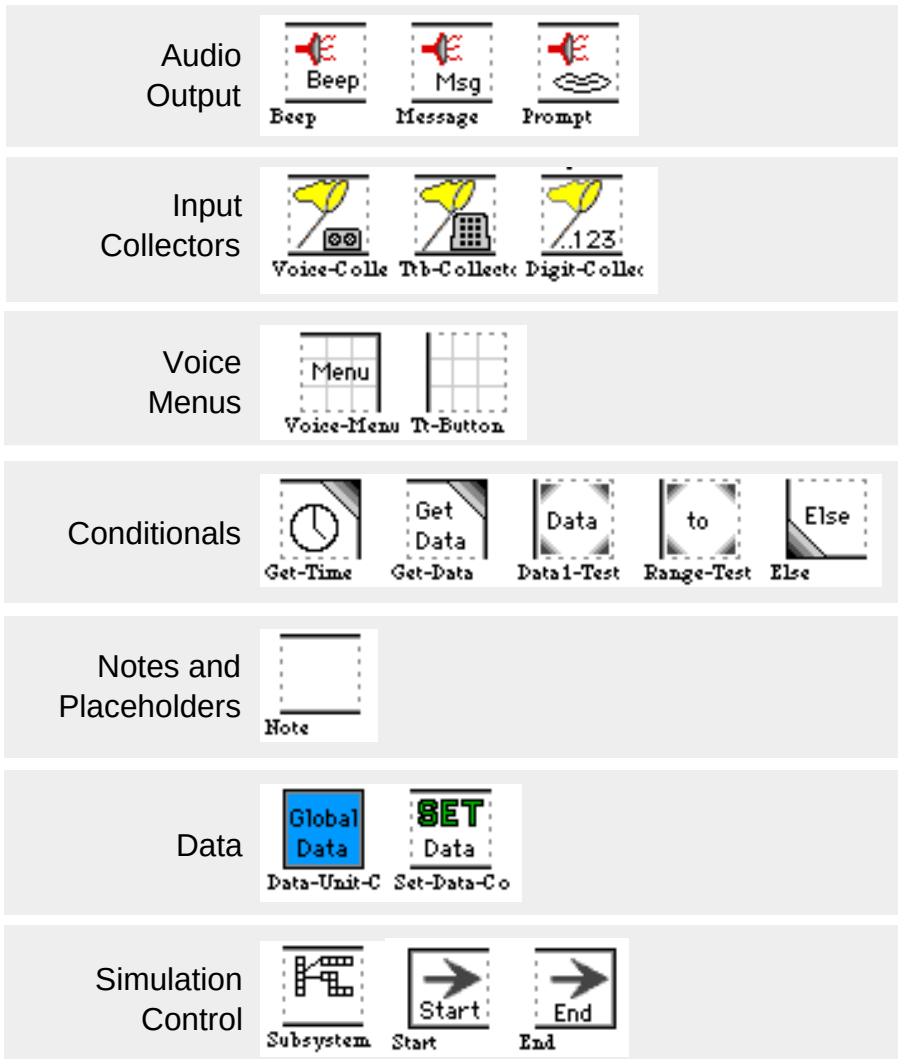

Figure 5.3: Design units provided by the VDDE gallery.

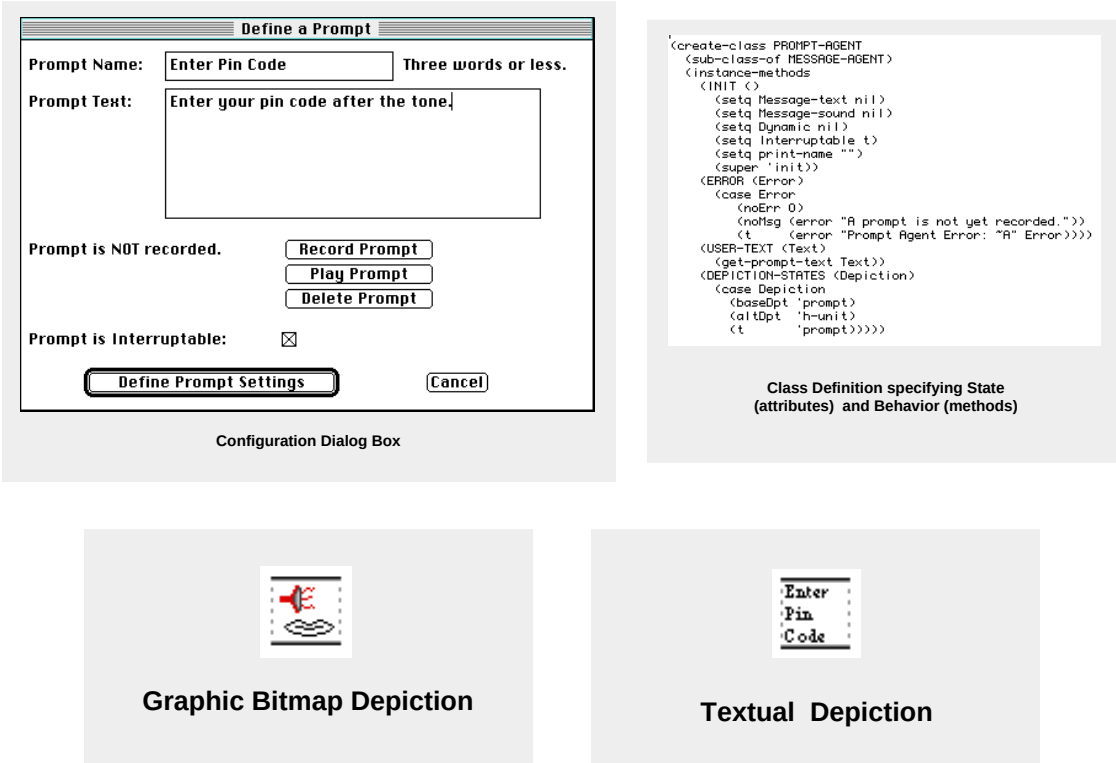

Figure 5.4. Parts of the prompt design unit.

A fundamental feature of Agentsheets is that worksheets have an underlying grid structure, consisting of unlabeled rows and columns. The grid structure guides the placement of design units. Communication between design units, i.e., the flow of information, is based on both relative and absolute spatial positions in the grid structure. In VDDE, these spatial communication relationships were used to create a simple, executable representational system. Three placement rules describe the order of execution flow when simulating the design representation (Figure 5.5). To create a design, designers assemble language components in the worksheet following these three rules. The borders of design units (i.e., dashed versus solid lines) also reflect these placement rules by indicating how design units can be legally joined.

The Critiquing Subsystem

The critiquing subsystem is described more thoroughly in [9, 10, 55, 78]; key points are reviewed here. As previously discussed, this subsystem was targeted at supporting designers to learn and apply

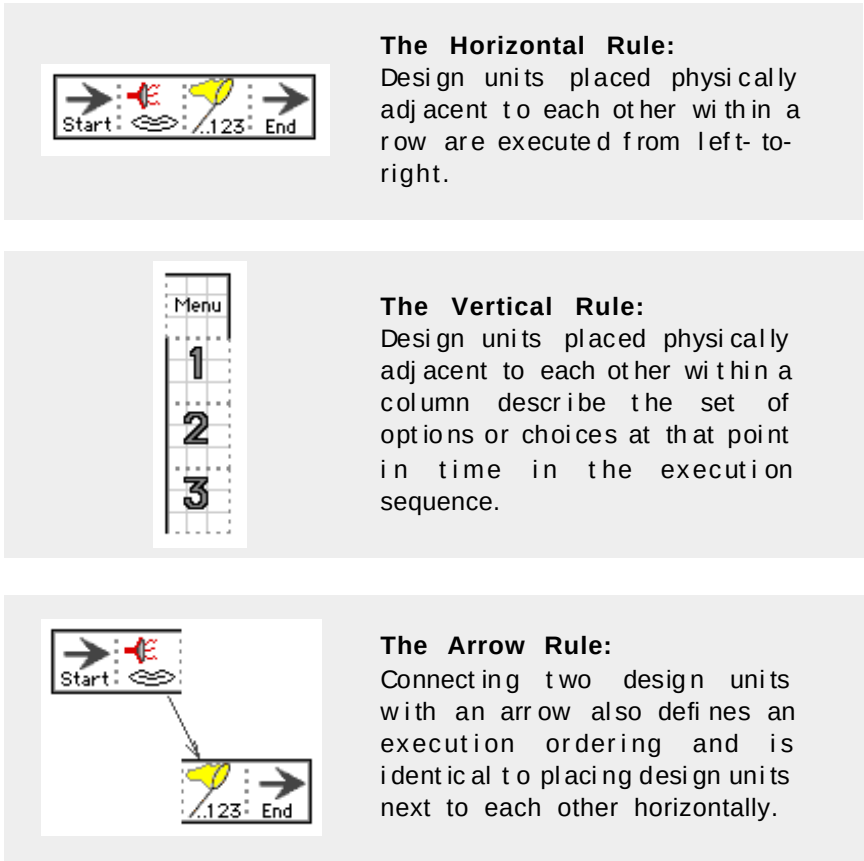

Figure 5.5: The Three Placement Rules in VDDE.

interface guidelines and to make trade-offs between the different guidelines and product consistency. The critiquing subsystem contains several DODE components: a knowledge-base of design rules, a hypermedia design rationale component, and a simple specification component.

Different from other DODEs, the knowledge-base is partitioned at the top level into four sets of critic rules that analyze design representations using different evaluation procedures. These different evaluation procedures and the content of the knowledge base were derived by analyzing existing user interface design guidelines and by analyzing the verbal protocols of designers during actual design sessions. Three sets of critic rules correspond to the regional, national, and international phone-based user interface standards. An analytical evaluation procedure is used to compare design solutions for

compliance with these three rule sets. A fourth rule set – the consistency set – uses a comparative evaluation procedure to compare two designs for inconsistencies. Each rule set is further partitioned by application type; e.g., voice mail, call answering, voice message delivery, and voice bulletin board.

The specification component provides a simple interface enabling designers to control the activities of the critiquing system. Using the specification interface, designers select and prioritize which rule sets to use during a design session; designers can selectively enable more specialized evaluative knowledge by specifying the type of application being developed. Designers can also control the intrusiveness of the critiquing system by adjusting the frequency with which designs are analyzed.

When the critiquing system is invoked, it analyzes the representation for compliance with all the enabled rule sets. When a possible violation is detected, a brief message signaling the violation is presented in a separate critic message window (see Figure 5.2). To help designers identify the source of a detected problem, each message is preceded by a symbol indicating the particular rule set the critic is part of. The system provides deeper explanations of critiquing messages by linking the messages with portions of the relevant user interface guidelines documented in the hypermedia knowledge-base. At any time, the designer can add new rationale, such as reasons why he or she is breaking a design guideline, to the hypermedia knowledge-base.

Architecture of VDDE

The final system architecture is shown in Figure 5.6. As previously discussed, VDDE is constructed using two tool substrates – Agentsheets and Hypercard [39, 123]. These are referred to as tool substrates because end-users do not interact with them directly in VDDE, but instead interact with specialized tools created using these substrates. Hypercard was used to create the specification and hypermedia design rationale components. Agentsheets was used to create the combined construction/simulation area. The two tool substrates are integrated at

a few points using the Apple event communication protocol provided by the Macintosh operating system. Communication is achieved by passing shared keywords back and forth. The receiving substrate will take a specific action depending upon the keyword received.

The combined construction/simulation area consists of several layers of domain models and tool extensions built on top of Agentsheets, the OPUS object-oriented system [93], and Macintosh Common Lisp (MCL) [1]. The domain model consists of two layers: the representational model layer and the domain object layer.

Above Agentsheets is the representational model layer defining the representation specific to VDDE; i.e., agent classes implementing the three placement rules. These classes are domain independent and not based on voice dialog design. This architecture evolved over time to facilitate experimentation with the representational system and to facilitate the introduction of new design units into the system.

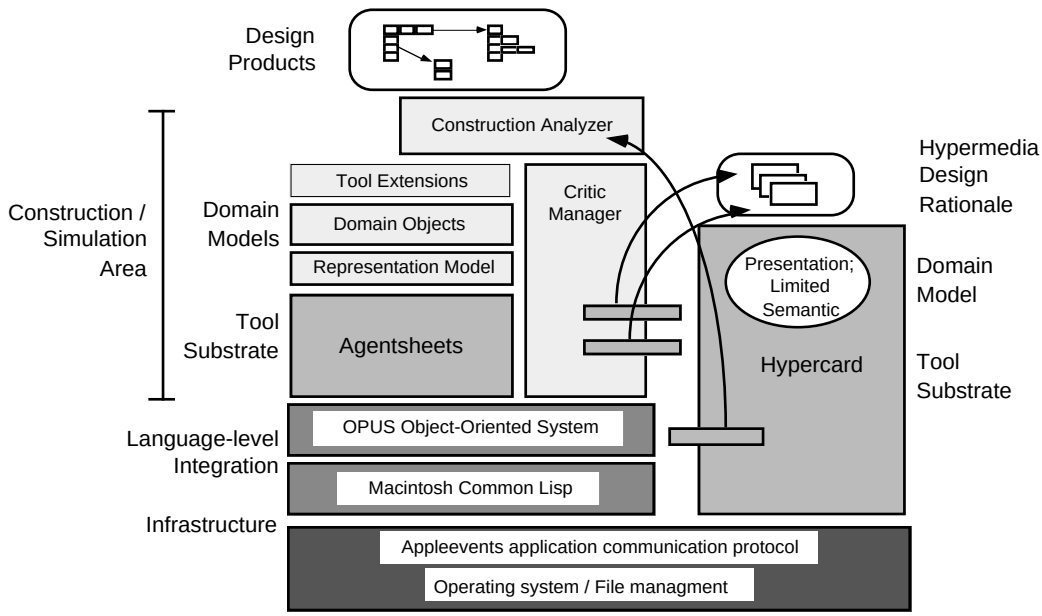

Figure 5.6: VDDE System Architecture

VDDE is built on two tool substrates – Agentsheets and Hypercard. The components of VDDE are integrated with each other using Apple events (shown here with rectangles and arrows).

Above the representation model layer is the domain objects layer. This layer includes classes defining the behavior of nested worksheets and classes defining the semantics and behavior of design units such as prompts, voice menus, beep tones, and digit collectors. Above the domain model layer is a tool extensions layer. This layer contains new or modified menu commands that have been added to the basic Agentsheets menu bar.

The remaining components belong to the critiquing subsystem. The critiquing system architecture is built on top of Hypercard, OPUS, Agentsheets, and VDDE's domain models. The specification and the hypermedia rationale components are built on top of Hypercard. These components communicate with the critic manager (Figure 5.6) using Apple event protocols.

The critic manager is the core of the critiquing system. It both contains the critic rules and coordinates which rules are enabled based on the current specification component settings. The critic manager consists of several, very large, specialized object classes created using OPUS. The construction analyzer controls the actual firing of critics and checks each enabled rule against the current worksheet (i.e., design product). The operation of the construction analyzer is based on objects and methods defined in the domain object and representation model layers. Additionally, several of the domain object classes in the domain model contain special hooks which trigger the activation of the construction analyzer. In essence, the critic manager and the construction analyzer are integrated at the language-level (i.e., programming code level) with the remainder of the VDDE system by the provision of special hooks in lower layers and by building on the content of lower layers. Thus, changes to these lower layers impact the operations of the construction analyzer and the critiquing system.

EXAMINING THE VDDE PROJECT TIMELINE

The VDDE project was concurrent with the voice dialog toolbelt story presented in Chapters 2 and 4. As the timeline shows (Figures 5.7-8), VDDE was in a continual state of change. Some of this change was inherent to the development process; i.e., our efforts to create an initial design environment. However, *the rate of change never subsided*. Two external factors contributed to the continual evolution of VDDE: innovations in the voice dialog *domain* and innovations in the Agentsheets *substrate*. This section will look at the interplay between these external factors and the evolution of VDDE.

Innovations in the Domain

Innovations in the domain arose both from within a particular design group and from the introduction of new design groups and new voice dialog products into the VDDE collaboration. What should be represented in VDDE and how it should be represented were in a continual state of flux. Basically, we were continually striving to strike a balance between *expressiveness* and *facility* [4, 65]. Expressiveness refers to the ability to state a solution given a particular set of abstractions in a language or representational system. Facility refers to how easy it is to state the solution given the abstractions of the language or representational system. Ideally, abstractions should be expressive enough to allow users to state solutions to commonly occurring problems, yet constrained enough to shield users from decisions and details they don't want to be bothered with.

To determine the appropriate level of abstractions, we employed a system design approach that was both problem-centered [65] and participatory [5, 20, 43, 44, 104]. A collaborative process was followed where voice dialog designers and system developers worked together to design and evolve domain-specific abstractions through use [99]. VDDE evolved through repeated attempts at solving real voice dialog design problems. Overall, its "domain-orientation" was the result of analyses of existing design tools and representations, analyses of voice dialog products in the marketplace, and collaborative design sessions

over the course of the project. These collaborative design sessions typically occupied a couple of hours and involved a voice dialog designer and the system developer sitting down next to each other and using VDDE to solve a real design task. As breakdowns in VDDE's abstractions were encountered, new design units were added or existing design units were modified or removed. In the following subsections, we will examine changes in the expressiveness, facility, and abstraction-level of the VDDE language as the result of these collaborative design sessions and other analysis activities.

Period of Domain Expansion: The Search for Expressiveness

Figure 5.7 illustrates VDDE's period of domain expansion. Version 1 shows the initial prototype, created in just a few hours. This prototype was shown to members of the voice messaging group to help gauge the suitability of the Agentsheets substrate for the future VDDE system. This prototype had no domain model and merely consisted of nodes that could be linked and could beep.

Version 2 shows the prototype after the first six months. By this time, the basic domain model had been created containing 12 design units and the three placement rules. The three placement rules were introduced to eliminate many of the arrows cluttering previous representations. In collaborative sessions, some designers used this early version to create simple voice dialog designs and found that VDDE's exclusive reliance on graphic bitmap depictions made the resulting design products unreadable. To address this problem, we introduced the notion of text overlays allowing all or parts of graphic depictions to be replaced with short text labels of one to three words. Also, this version did not yet support audio recordings; instead, it relied on MacinTalk to voice the prompts and messages encountered during design simulation. The voice messaging designers noted that they had tried MacinTalk unsuccessfully before; they had found that the sound quality was not high enough to show to potential users or customers.

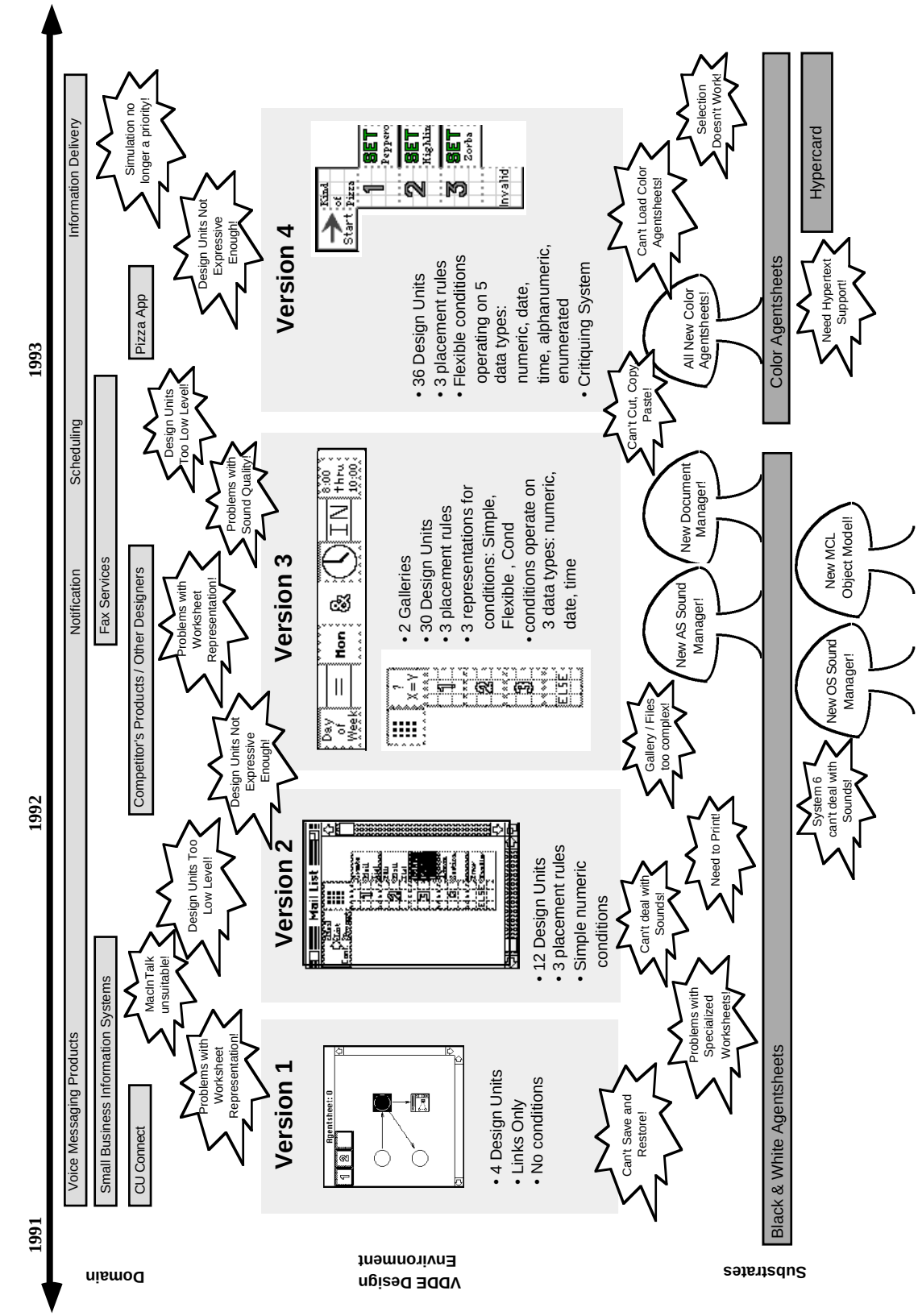

Figure 5.7: VDDE project timeline during the period of domain expansion.

Version 3 shows the design environment by the end of 1992. The VDDE system had exploded to include multiple galleries with approximately 30 design units in each gallery. Each gallery was created to experiment with different ways of representing conditionals. An analysis of competitors' voice dialog products, a new collaboration with a fax services group, and new design contracts for notification and scheduling features in the voice messaging group had all highlighted the system's insufficient expressivity in this area. We added new design units addressing different data types (e.g., alphanumeric, date, and time) and experimented with new conditional representations (called "cond", "flexible", and "simple"). Additionally, we were now using third party software and recording devices to create high-quality audio prompts. While the resulting sound quality was excellent, the use of these applications was cumbersome since they were completely unintegrated with VDDE. Designers had to save each recording in a separate, named sound file and then input this file name into the associated design unit in VDDE. Unfortunately, there was no alternative at the time: the Macintosh operating system at that time (version 6) did not provide any built-in support for recording sounds, so we had to use these third-party applications and the third-party applications being used were not extensible or integratable.

By version 4, we had settled on the flexible conditional representation. This choice was based on the results of extensive programming walkthroughs [3, 4] performed by eight groups of interface designers comparing the different forms of conditionals [118]. There were now 36 design units in the gallery; six new ones had been introduced to support simulating designs operating on enumerated data types.

Overall, two factors drove the expansion that characterized the first four versions of VDDE: the desire for *amortization* and the desire to support design *simulation*. As noted by Prieto-Diaz and Arango, to be cost effective, the effort of building a domain-oriented system must be amortized across a community [2]. The desire for amortization, or broader applicability, inspired us to understand the design needs of

multiple design groups and to analyze competitor's products and other types of voice dialog products. This favored emphasizing expressiveness in the VDDE language as we modified the contents of the gallery (i.e., the design units) to be able to state a wider range of design solutions. In our case, expressiveness corresponded to a proliferation in the number of design units.

As each new group or new product entered the collaboration, we minimally had to introduce new design units into the gallery to cope with the new demands placed on the system. Sometimes, even the underlying representational system, particularly in the area of conditional actions, was not expressive enough and it too had to be changed.

Another factor contributing to the drive towards more expressiveness was the goal of supporting design simulation. At the start of the project, the designers wanted to bypass using existing simulation languages when building prototypes for user testing and the marketing groups. These languages were essentially general-purpose programming languages enriched with special constructs for handling phone hardware and audio recordings. Robust simulations could be built using these languages but they were hard to learn and hard to use. Many new designers did not know the simulation languages and were unwilling to learn them.

So in the first half of the project, a central design goal was being able to create simulations supporting "full-blooded" action. Full-blooded simulations are detailed simulations that provide realistic use experiences [16]. In voice dialog design, full-blooded action means providing functionality such as driving phone hardware, high quality sound, and the ability to interrupt sounds with key presses. For a design to be simulatable or executable in a full-blooded sense, it must be explicitly and precisely stated. In effect, this requires the representational system to also serve as a programming language. Thus, the goal of realistic simulation also led to more design units, with each design unit having a more narrowly defined definition. As

shown in Figure 5.3, the resulting VDDE gallery provided multiple, specialized versions of many of the basic types of design units found in the existing flow chart representation.

Period of Domain Shrinking: The Search for Facility

A competing factor began to counter the drive towards greater expressiveness and promote the drive towards facility instead. This factor was the desire to support upstream design activities; i.e., more conceptual design and problem-solving activities as opposed to more downstream, detailed specification activities.

By now, in our collaborative sessions, we were trying to use VDDE to create designs for contracts currently being handled in the voice messaging group; i.e., real design tasks. In a series of design sessions, we saw that the expressiveness of the design units in the gallery significantly changed the design task. Levels of detail that used to be handled in downstream design activities, or even not at all, were now moved to the forefront of upstream activities. For the design to be simulatable, designers had to specify detailed conditional actions and to define operations on underlying data items such as schedules or mailboxes. Using flow charts and text design representations, conditional actions were not specified until further downstream in the process. Operations on underlying data items were often not expressed at all but instead left implicit; the vendor implementing the application dealt with these issues. Because of this enforced level of detail and explicitness, designers were often not able to sit down and design using VDDE; they first constructed a higher-level representation and then transcribed this representation into VDDE.

These design sessions indicated that the abstraction level of VDDE's design units was too low, oriented more towards supporting detailed design simulation than supporting upstream design construction. To remedy this situation, we began to aggregate and eliminate existing design units to simplify the gallery; i.e., VDDE began to undergo a period of domain shrinking (Figure 5.8).

Figure 5.8: VDDE project timeline during the period of domain shrinking.

Version 5 shows VDDE after a period of aggregation. Many of the design units representing atomic actions had been combined to create higher-level design units (such as combining messages and touch tone buttons to create voice menu units). Additionally, the initial design goal of supporting full-blooded simulation was no longer a priority. By that time, the voice messaging group had somewhat modified its practices so that specialized simulation builders located in the group developed all of the prototypes; thus designers did not need to learn or use the simulation languages. This shift in practices led us to shift VDDE design goals towards the sketchpad concept. Since the environment no longer needed to express complex conditional actions, we reverted back to the "simple form" of conditional representation and this eliminated several more design units from the gallery. To help designers cope with the, still large, number of design units in the gallery, the notion of visual depiction themes were introduced. Visual themes were used to indicate design units all belonging to a similar functional class, such as the theme of butterfly nets to distinguish input collectors. We continued to eliminate functionality, mostly in the area of conditional operations, and eventually reduced the number of design units to 18 (Version 6).

However, later use sessions indicated that we still had not gone far enough in simplifying the VDDE language. The design representation was still oriented towards expressiveness and programmability; there were still too many low-level design units. As we saw in the voice dialog stories in Chapters 2 and 4, designers seemed to prefer between four and eight domain abstractions; VDDE had eighteen. However, even though VDDE had too many design units, it still couldn't express all the concepts that needed to be expressed. It was a catch-22 situation.

Our solution to the expressiveness problem was to introduce the Note design unit (Figure 5.3), which was a textual placeholder that did nothing. As one designer said during a later design session, "I've fit this on a number of other designs even, in trying to use more conventional flow charting, *where there's just certain things that don't lend themselves to the representation you're using.* .... And actually,

well I can see why I can get really carried away with this notes stuff. I like this. I want to do one more." [italics added for emphasis]

Innovations in the Substrate

What is particularly interesting was the considerable impact innovations in the Agentsheets substrate had on the evolution of VDDE. As Figures 5.7-8 show, innovations were periodically "mushrooming up" from below. Some of these innovations resulted from changes in Agentsheets; other innovations stemmed from changes to MCL and even the underlying operating system.

Some of the Agentsheets' innovations were in response to breakdowns identified in the VDDE system and were crucial for the success of VDDE. For example, innovations in the sound manager were required to meet the simulation needs of the project. Sometimes, innovations were introduced for the benefit of other Agentsheets applications (e.g., Color Agentsheets, Windoids, Toolbars). Several of these innovations did not just impact implementation issues; they represented major changes to the overall look, interface, and functionality of the VDDE system.

In summary, the evolution of VDDE was driven from two ends: mostly from the domain above, but also from the substrate below. While it is not surprising that both the domain and the substrate exhibited innovations, what is surprising is both the rate of change and the extent of the changes in both. Between the two of them, the VDDE system was continually undergoing fairly significant modifications to simply stay abreast of their innovations.

ANALYZING THE VDDE PROJECT

In this section, we'll examine the VDDE project from a domain construction perspective (i.e., support for use and support for evolution in the form of elaboration and modification processes). Chapter 4 analyzed commercial products, which the designers were using on their own. This chapter analyzes a research effort during the

course of its collaborative development with some of the same designers. Thus, this section differs from the analogous section in Chapter 4 by analyzing what *both the designers and the system developers* were able to do with VDDE prototypes.

During this analysis, I will also look at the type and extent of modifications performed to the system over the course of the project and point out places where the special skills of the developers were required. In the discussion, I will try to make educated judgments as to where those skills might not be necessary in a more complete VDDE product; i.e., which skills were required for initial development purposes versus the skills required to normally use and evolve the system.

Support for Use

Table 5.1 summarizes VDDE's support for use processes. Overall, VDDE provided mixed results at supporting designers to construct representations using domain vocabularies. From the project timeline discussion, it is clear that the level of abstraction of the domain vocabularies was too low level to support the creation of upstream design representations.

Table 5.1: VDDE support for USE.

Support for USE	**VDDE**
• full DL spectrum supported?	explicit / formal only
• constructing	downstream favored
• evaluating - inspecting - analyzing - relating - envisioning & experiencing	Yes, with limitations - little support for inspection - analyzing supported in unexpected ways (anticipations) - experiencing partially supported
• iterating	No

However, in terms of supporting voice dialog designers to quickly create their own simulations, the environment was quite successful. Many designers who were unable to create their own simulations using existing languages were able to create simulations in VDDE. Additionally, these simulations only took "hours," rather than the customary "days" to build. In one informal bake-off between an existing simulation language and VDDE, two designers constructed simulations for the same design. It took one designer 3 hours to construct the VDDE simulation; it took the designer using the simulation language the better part of a week. Unfortunately supporting these two processes – upstream design and simulation – seemed to be in opposition and very difficult to do with a single representational system.

VDDE's support for design evaluation was quite positive with some limitations. These designers engaged in three basic kinds of evaluation: visual inspection, envisioning, and analyzing designs with respect to design goals and guidelines. The observations presented in the following paragraphs are based on workplace observations and analysis of videotaped design sessions of designers working both with and without VDDE. More information on these results can be found in [9, 10].

The limitation stemmed from VDDE's lack of support for visual inspection. Basically, the representational system provided by VDDE did not support visual inspection processes very well. In early use situations, designers complained that they couldn't tell what design they were working on when looking at a worksheet; their design could be any phone-based interface. This was due to the use of graphic bitmap depictions that emphasized the functionality or behavior of a class of design units rather than the specific content of any instance. For example, the lips in the prompt design unit indicated that its function was to play recorded voice input but provided no information concerning the content of the specific prompt. We tried to rectify this problem by modifying the depictions to include some textual

information but were limited with how far we could go with this approach given the affordances of the substrate at that time.

Additionally, these problems with visual inspection got significantly worse as designs increased in size, even more so than in the existing flow chart representations. In part, this was also due to problems designers had in laying out designs using the three placement rules. As noted by Petre, over time experts evolve secondary notations guiding their production of design representations that make resulting design products easier to inspect and interpret [86]. While the three placement rules were simple to follow, quite a bit of learning and use seemed to be required to derive secondary notations for making these representations more readable.

It should be noted that recent versions of the substrate (released since the end of the VDDE project) now provide significantly more support for larger depictions containing text and subparts. It might now be possible to create a representational system that overcomes the "lack of content in depiction" inspection problem and create depictions containing the full text of prompts and messages. However, the layout problems with the three rules raises more general questions concerning the design of information representations. The lesson we learned was that usability of an information representation is a factor of both the explicit and the implicit rules of form and both need to be considered when creating representational systems.

VDDE's simulation facility did support envisioning the future audio product. Several designers detected design defects in the audio portions of their design using the simulation features. Interestingly, the act of recording triggered designers to notice potential problems just as much as hearing the simulation did. However, sometimes the effort associated with recording audio messages discouraged many designers from using the simulation facility. Some of the perceived effort stemmed from the system's multiple step interface for recording prompts (which has since been redesigned to be more streamlined).

Other aspects of the perceived effort may have stemmed from the extra explicitness required to record prompts. Most designers used the "traditional" practice of voicing (speaking out loud to themselves) audio messages during design. VDDE's simulation features did not appear to replace this method, but rather to augment it. Often designers using VDDE's simulation component still did the "traditional" voicing before creating recordings. With the voicing method, designers often did not actually say the entire, exact audio message, but instead an approximation of the message. When using VDDE's simulation feature, they seemed to always try to record the exact message to be heard in the final audio product. Sometimes, this took two or three attempts to get the recording and phrasing right. This extra explicitness sometimes pointed out problems in their design; however, it may also have contributed to the perception that taking advantage of the simulation feature required extra work. Further evaluation of the simulation system needs to be performed to gauge if the perception of effort has changed as a result of the new streamlined design.

VDDE's critiquing system was expressly created to support evaluation activities such as analyzing design solutions for compliance with design rules. Observations and empirical studies indicate that the critiquing system did support such evaluation activities but often in surprising ways. For experienced designers, the critiquing system had little direct effect on design products; i.e., rarely did designers modify either the solution or the specification in response to critic messages. This seemed to result from two factors. First, experienced designers rarely broke design rules, thus the critiquing system was often inactive during design sessions. Second, designers *anticipated* critiquing system activity. Sometimes these anticipations caused designers to change their design activities in advance and thus prevent the anticipated critic from firing. Other times, by the time the anticipated critic fired, designers had already decided to not follow the critic rule anyway and so did not modify their design products. Analysis of protocols and subsequent interviews with designers indicated that the awareness of the critiquing system being present did lead experienced designers to deeply reflect on their design actions and particularly, why they were

choosing to break various design rules. Thus, the critiquing system did enhance evaluations but in an unexpected and indirect way. More detailed analyses of the critiquing system and its differential effects based on the domain-skill level of designers can be found in [10].

VDDE did not affect the designers ability to iterate their design as described in Chapters 2 and 4. The designers still constructed the flow chart and table representations; when used, VDDE served to supplement these activities. Since these representations were totally outside the scope of the system, VDDE did not help with the original maintenance problem and possibly contributed more to the problem by introducing a third design representation to maintain.

Support for Elaboration

Table 5.2 summarizes VDDE's support for elaboration processes. As illustrated in Figures 5.8-9, we (voice dialog designers *and* system developers) were able to articulate new and refine existing domain vocabularies (in the form of design units) continually throughout the project. We were also able to extend the domain model to include object behaviors (to support simulation) and tool extensions (new menu commands) were added to the Agentsheets substrate. However, there was a wide range in the effort and special knowledge required to create and modify different design units.

For some design units, the effort was very small, requiring only slight extensions to existing functionality already provided in VDDE. Some of these design units could be created in minutes, on-the-fly, during actual use sessions [99]. While these real-time extensions were, for the most part, successful, there were also a few difficulties in the area of smoothly reflecting changes in open design products. If the extensions involved adding new attributes or methods to an existing design unit class, then open worksheets needed to be closed and reopened for the changes to affect the design units already in the worksheet. It is conceivable that technically knowledgeable designers, such as local developers [36], could make extensions such as these without the presence of a system developer.

Table 5.2: VDDE support for ELABORATION.

Support for ELABORATION	VDDE
• DL transitions supported?	explicit to formal only
• graphical refinement	Limited. - Can Require Technical Knowledge - Class level refinements only
• naming - name / look mappings	Limited. - Requires Technical Knowledge
• object refinement - whole / part mappings	Yes, with limitations - Additions and deletions problematic for design products
• establishing relationships - correspondence - equality - transformation - heuristic	Possible, but difficult - No Direct Manipulation - Collection of ad hoc mechanisms - Low-level programming
• extension - object behaviors - tool extensions	Possible, but difficult - object behaviors - tool extensions - Requires Programming Knowledge

Other design units required moderate programming effort lasting several days. These units typically transcended the existing functionality of VDDE and required extensive Agentsheets programming (e.g., introducing enumerated types or Digit Collectors). Still other design units required extensive programming effort to create, lasting from several weeks to dragging out over the course of months. These units needed functionality that transcended the Agentsheets substrate and required extensive MCL programming (e.g., configuration methods and dialog boxes for prompts, phrases and voice menus). In some cases, design units required functionality not available in the current release of the operating system and we had to wait for future releases for the desired functionality (e.g., waiting for new sound managers).

On the one hand, this level of effort might be attributable to "development pains" that would not normally appear in a more complete VDDE. On the other hand, maybe this represents the normal state of affairs in innovative domains. I'm inclined to think the answer is somewhere in the middle. Certainly as our architecture evolved, we were able to make additions and changes more smoothly and easily. However, even towards the end of the project, occasional needs would arise that required extensive programming changes to VDDE. Perhaps some of these changes could be carried out by technically knowledgeable designers using new end-user programming approaches such as Visual AgenTalk [97]. However, I think it would be overly optimistic to believe that system developers would not be needed on a periodic basis to make the necessary changes.

As demonstrated by the critiquing system, we were able to establish heuristic relationships across representations. The critics in the system represented heuristic relationships between parts of a design solution in the worksheet (in Agentsheets) and parts of the design rationale (in Hypercard). It would require extensive programming knowledge, and probably developer support, to add new or modify existing critics rules as the system currently stands. No provisions are made for establishing other forms of relationships.

For the most part, the supported elaboration processes emphasized making the explicit to formal design language transition. There was limited support for the tacit end of the design language spectrum; i.e., *in situ* graphical refinement and naming. As discussed in Chapter 2, supporting this part of the spectrum is important for design processes in innovative domains where designers continually articulate and refine domain vocabularies *during* design. System supporting this end of the spectrum cannot require the continual presence of system developers for making minor changes to the design language. Ideally, systems would also not require the designer to step too far out of the design process and engage in a seemingly separate modification activity to perform simple presentation-level changes.

On the one hand, Agentsheets does support some level of tacit exploration when first creating design units. Any new depiction in the gallery can immediately be used on the worksheet without having to formally define an associated class; these depictions get automatically associated with a default class.

On the other hand, the support for tacit exploration is limited when it comes to modifying existing depictions, particularly in VDDE. VDDE itself limited the extent of graphical refinement possible with its use of textual overlays. These overlay positions were built into design units and any modifications to depictions needed to take these positions into account. Additionally, depictions needed to be created with particular color masks for the overlays to be selectable. Between these two factors, considerable technical knowledge and thus, system developer support, is required to graphically refine depictions in VDDE.

Additionally, in Chapter 2 we saw that graphical refinements tended to be performed *in situ* on specific instances of domain vocabularies embedded in the representation. Later on, some of the instance refinements were incorporated into the general class of item. This kind of instance level refinement is not possible given the object-oriented model VDDE/Agentsheets is based on, which provides for class-level refinements only.

Support for tacit exploration in the area of refining existing object names is also somewhat limited. This is because the names in the Agentsheets' gallery provide look/behavior mappings in addition to name/look mappings. For all design units, there is a mapping between the depiction name and its associated object class. Modifying the depiction name requires modifying this mapping or the association between look and behavior will be lost. Thus, while it is possible to rename design units in the Agentsheets' gallery, this action is intended primarily for system developers and can lead to rendering parts of the system inoperable if done unawares. While maintaining this mapping does not require great programming skill, it does nevertheless require

awareness of the content of the underlying object hierarchy and such refinements may require the presence of a system developer.

Support for Modification

Table 5.3 summarizes VDDE's support for modification processes. While we were able to experiment with and create new representational systems, both activities required substrate evaluation and programming knowledge. Thus, it is unlikely that designers could experiment with and create new representations without extensive system developer support.

Table 5.3: VDDE support for MODIFICATION.

Support for MODIFICATION	VDDE
• experimentation and creation	Limited • Requires Programming Knowledge • Requires Substrate Evaluation Skills
• introduction - augmentation - replacement	No
• retrofitting	No. - Even manual very limited.

Two representational systems were created during the project. The core representational system was the schematic flow chart-like representation used in the worksheets. Halfway through the project, a second representational system – the design rationale component – was created to augment the existing environment. In both cases, before a new representation could be created, a substrate had to be located and evaluated for suitability. When evaluating substrates, we considered criteria such as extensibility, affordances, and code stability. During the first few months of the project, we looked for a suitable substrate and, after reviewing several, selected Agentsheets. When the critiquing system was being investigated, previous experience with Hypercard led us to quickly choose that particular substrate.

The two substrates differed in their ability to quickly create and modify a prototype. Using Agentsheets, we had a rapid start-up time. We were able to get an initial prototype going in a matter of hours and a significant design environment prototype going in four months [117]. Also, once created, Agentsheets made it significantly easier to perform certain modifications to the representational system such as experimenting with conditional representations and various placement rules. Once the Hypercard stack was created, it was difficult to make modifications without having to reorganize significant amounts of information. Other projects have reported difficulties modifying Hypercard prototypes [7, 71], noting that its "missing object-oriented features made it harder to create and modify domain-specific building blocks" page 63, [71].

As the system developed further, we found there was an interdependence between the two substrates that reduced our flexibility. Once the critiquing system was created, our flexibility to modify the representational system and even the design units was significantly reduced. It was very difficult to modify the representational system used in the worksheet because the critiquing system's had encoded in its construction analyzer the expected spatial relationships between the design units. It was difficult to modify some of the design units because the hooks to activate the critiquing system depended on both the existence of and the specific internal structure of certain design units (such as the start, touch-tone button, and voice menu units). Introducing a new representational system into VDDE (to replace) the existing worksheet representation (i.e., replacing the representation model layer) would have rendered the entire critiquing system inoperable. It should be noted that when the critiquing system was created, we did not yet understand the critical role that elaboration and modification processes played in this domain and thus, these issues were not considered in the subsystems design.

VDDE fell short at supporting retrofitting activities. Once the domain model was changed, many designs created with previous

versions of VDDE were now unreadable. Once a design unit class or even a design unit attribute was removed, existing designs could no longer be opened (unless the saved worksheet LISP representation was edited directly). There was no support at all for retrofitting old designs to new design languages other than to completely recreate the design from scratch manually. The only way to ensure that an old design was not lost completely was to print it out and save the hardcopy. This was exacerbated by the process for restoring worksheets. If a class was encountered during restoration which no longer existed, then none of the design could be restored. This was the case even if the encountered class corresponded to one undefined design unit out of one thousand other defined design units.

This problem could be alleviated if only non-monotonic additions are made to the object hierarchy. However, given the evolutionary development approach used in this project, following such a strategy would have resulted in a chaotic and difficult to maintain object hierarchy and would have hindered our ability to elaborate and experiment. Very occasionally, there would even be changes at the substrate level where a class or method would be removed; thus such conditions are sometimes out of the control of the design environment developer. Basically, support was needed for managing different versions of design languages.

REFLECTING ON THE GENERAL DODE MODEL

In the previous chapter, the bulk of the domain construction analysis centered on the general Toolbelt model. In the case of DODEs, we are limited in our ability to analyze the general model for two reasons. First, only a single DODE has been directly experienced as part of this research. Second, while previous DODEs have been created, these environments were focused on exploring theoretical ideas and there is little previous experience with use of DODEs by professional designers in workplace settings that we can generalize over. As a result, in this chapter, the bulk of the domain construction analysis centers on VDDE rather than the general DODE model.

However, it is still worthwhile to speculate how VDDE experiences might inform the general model. In this section, we will do just that and, where possible, compare VDDE experiences with other DODE efforts. The following subsections consider each of the three main areas in the domain construction framework – process support, technical aspects of the computational model, and special skills and knowledge that might be required by the model.

Support for Domain Construction Processes

Recall from the introduction our initial intuitions that domain-oriented systems provide good support for use but could be less flexible for accommodating change (i.e., elaboration and modification). After analyzing VDDE, it seems that these intuitions are only partially correct on both counts.

In VDDE, we saw that the presence of a formal model did provide support for use activities such as design simulation and evaluation. However, these benefits were with some costs, namely in the area of excessive explicitness that adversely affected the system's ability to support design construction. It proved to be a challenging system design task to create an environment providing these benefits without stumbling into the "tyranny of the explicit" [57] danger zone. Some of these problems could be VDDE-specific, stemming from our design decision to use a single representation for both construction and simulation. However, some aspects of the explicitness problem are more general; it will always require a certain level of explicitness for computers to be able to parse and execute design representations as necessary for this type of knowledge-based support.

We also saw that VDDE exhibited a surprising amount of flexibility, particularly in the area of design language elaboration processes. By building on top of the Agentsheets substrate and layering the internal domain model, VDDE was able to accommodate significant evolutionary change in domain vocabularies and to some extent, even the representational system. However, true to our intuitions, the

presence of the domain model did hinder some aspects of our flexibility. Once the critiquing system was in place, the range of possible design language elaborations was reduced. As a whole, the system fell short at supporting design language modification processes. It is interesting to consider how much of this inflexibility is inherently due to the domain-orientation and how much is due to the specific architecture and object model adopted in the VDDE project. These issues will be considered further in the following section and in more detail in Chapter 6.

In general, many of VDDE's problems stemmed from the emphasis of the system on supporting transitions from explicit to formal design languages only. The tacit to explicit end of the spectrum was completely unsupported. There were few elaborations and modifications that designers could perform without the assistance of system developers. In a dynamic and innovative domain such as voice dialog design, this is an insurmountable bottleneck.

This lack of support for tacit to explicit transformations, particularly in the area of the construction component, seems to apply to the general DODE model. Furthermore, it appears to stem from the model's current reliance on the construction kit concept to instantiate the design construction component.

Previous DODEs supplied end-user modifiablity support tools (MODIFIER) to help designers modify the construction kit [26, 38]. However, MODIFIER did not support making tacit to explicit transitions. The approach used in MODIFIER required designers to completely specify the formal model of a domain object at the time of introduction; thus MODIFIER only supported explicit to formal transitions.

As discovered by VDDE and other DODE projects [107], end users seems to be unable and / or unwilling to formalize knowledge at the time of input; i.e., they both need and prefer to input information in more tacit (informal) forms. As demonstrated by our experiences in

VDDE with the Note design unit, occasionally abstractions arise in design that fall outside of the current representational system. In many of these cases, it would be impossible to formalize these abstractions at the time of input because the designer's understanding of their form and content is still at an implicit, tacit level.

In the DODE argumentation component, this inspired *in situ* incremental formalization approaches [107] whereby natural language annotations and messages are gradually formalized into structured design rationale and underlying domain objects. Based on the experiences in VDDE, such an approach seems advisable for the construction component also; the emphasis should be more on providing designers with a "construction kit for creating construction kits" instead of the end product. As discussed in Chapter 2, designers often start with generic graphic objects and incrementally refine them over time to be more explicit and detailed in their definition. Based on these observations, it seems desirable for meta-construction kits to provide some support for in situ graphical refinements of form and possibly even content as well. In Chapter 7, these speculations will be further explored.

Technical Aspects of the Computational Model

When analyzing computational models and architectures, Garlan and Shaw [37] note that it is crucial to identify the model's invariants; i.e., the aspects that are the same across all instantiations. In the case of DODEs, this is difficult to do at the architectural level. This variance in architectures is to be expected though considering that DODEs are still an active research area.

To date, the DODEs constructed fall into two main architectural categories: component- or substrate-based architectures and shared object model architectures (Figure 5.9). In this section, these architectures will be considered with respect to their potential impact on supporting design language elaboration and modification processes. Specifically the analysis will consider the computational model parts

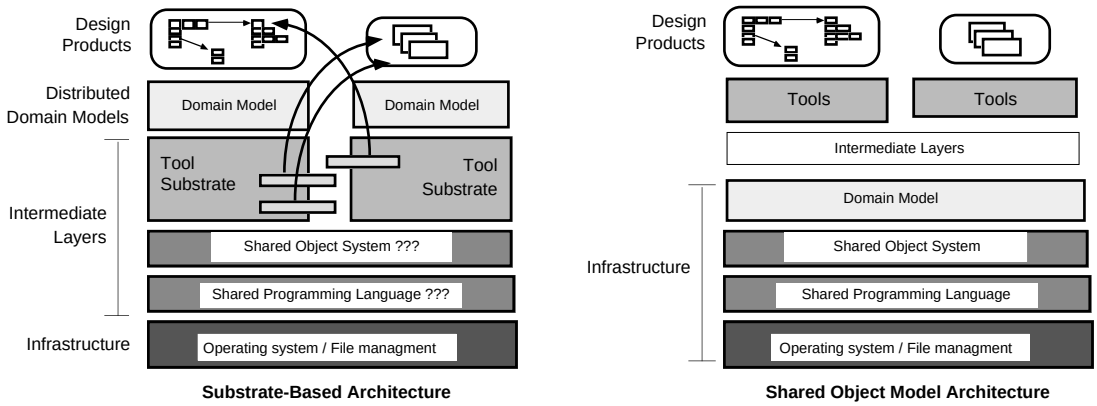

Figure 5.9: The two main DODE architectures.

introduced in Chapter 3 – object models, domain models, tools and substrates, and integration services.

There have been numerous DODEs built on a component-based architecture (JANUS [31], KID [76], Indy [91]). VDDE and ProNet [115] are built on a substrate-based architecture. The key difference is whether or not the tool providing the particular DODE component, such as the construction or specification component, is built on top of a substrate such as Agentsheets or implemented from scratch. In VDDE, it seemed that the substrate approach provided much of the system's flexibility. It remains to be seen if the success of the Agentsheets substrate in this regard can be replicated in other substrates. Probably its support for incremental refinement can be replicated, but more challenging is its "construction paradigm" approach [94] for reusing high-level behaviors and interfaces.

Additionally, another factor contributing to VDDE's flexibility with respect to design language elaboration processes was its layered domain model. Particularly, its separation of the representational model from the domain object model enabled new objects to be introduced with less effort and it enabled the representation to evolve without affecting already defined objects. This separation is similar to the approach used by Girgensohn to support end-user modifiability in JANUS [38]. Girgensohn encoded the semantics of the floor plan representation in terms of spatial relations in a separate underlying layer. All relationships between domain objects were expressed in terms of these

spatial relations. Thus, the semantics of the floor plan representation could undergo some level of change without requiring existing domain objects to be modified.

In component- or substrate- based architectures (Figure 5.9, left side), key similarities are distributed domain models and reliance on point-to-point integration mechanisms (i.e., components integrate with each other rather than the infrastructure [24]). In this respect, they are similar to the coalition architecture used by the Toolbelt Model (Chapter 4). The parts of the domain model essential to each particular component in the multi-faceted architecture are localized and usually cannot be accessed by other DODE components. Since none of the domain model is shared between components, this results in much reliance on point-to-point, control-oriented integration mechanisms. In general, these integration mechanisms make the resulting system very inflexible. An example is how the critiquing system in VDDE made it difficult to elaborate and modify the domain vocabulary and representational system employed in the construction component.

Some of this inflexibility may be VDDE-specific. However, it seems reasonable to infer that this architectural approach in general will put a high cost on modification processes such as the introduction of new representational systems. If an existing representational system and its associated tool or substrate is replaced, then the domain model associated with that component will probably not be reusable. Furthermore, all links or integration points will also probably be rendered invalid and have to be redone to work with the new component.

In the DODEs built to date with this style of architecture, the presence of a shared object model and shared programming language has been variable; thus the possibility for data-oriented integration has not really been present. In VDDE and ProNet, there was no shared data/object model or shared programming language; each substrate provided its own. Whether this lack of sharing is endemic to the substrate model or not, remains to be seen in future DODEs based on

this model. Girgensohn claimed that some level of object model sharing was required to support end-user modifiability and he introduced a shared, domain-independent, object layer into JANUS.

Other DODEs (Hermes [111], X-Network [106], and EVA-Service [84]) have gone further and proposed that the domain model should be shared across all design environment components. These environments are based on a shared object model architecture (Figure 5.9, right side). In this architecture, much of the functionality is provided in shared infrastructure services instead of in individual tools or components. One such shared service is an integrated hypermedia-based object model. In the DODEs created, the exact features of this object model vary from system to system. These environments used the shared object model to create a shared domain model used by all DODE components. As a result, these environments provide both data-oriented and control-oriented integration services. The control-oriented integration is provided by intermediate layers of computed nodes or queries [52, 106, 111]. Proponents of this architecture claim that it results in better component integration and that the use of a hypermedia object model provides the system with greater flexibility [106, 111].

Experiences with X-Network indicate that this architecture can indeed accommodate flexibility in terms of elaboration processes [107]. Unfortunately, little direct data is available on the flexibility of this architecture in terms of modification processes. However, there are some experiences with this architectural model in the related area of CASE (computer aided software engineering) tools. Feiler and Wallnau note that this style of integrated architecture was pursued in the 1980's with first-generation CASE tools [24]. In the CASE arena, this architectural style was supplanted with coalition architectures because it proved unable to keep pace with the rate of change and it did not provide the expected level of value and integration. It seemed that the ideal of a completely shared domain model was never actually achieved; over time, tools usually ended up augmenting the shared domain model with their own local domain model to service their

special needs. Feiler and Wallnau note that some of these problems may have stemmed from the immaturity of object-oriented technology at that time.

In summary, it is clear that architecture plays an important role in an environment's ability to accommodate evolutionary change. With respect to elaboration processes, factors such as substrates and the separation of representational models from domain object models appear to positively contribute to a system's flexibility. It is also clear the integration services deeply affect a model's ability to accommodate modification processes; however, it is not at all clear what types of integration services are best. In Chapter 6, we will revisit these architectural issues when we compare Toolbelt and DODE approaches.

New Practices, Skills and Knowledge Required

Table 5.4 lists the new practices, skills, and knowledge required by both designers and system developers in the DODE model. Not many new skills are required by designers. However, due to the continual rate of change in dynamic domains and the current DODE model's lack of support for tacit (informal) design language extensions, some designers must be willing and able to engage in end-user programming as necessary to elaborate their design language. The particular language or extension mechanisms provided by the model is still an open research issue. Several new approaches to end-user programming are being investigated in the Agentsheets substrate [96, 97]. These approaches offer much promise with respect to empowering designers to modify and extend the behaviors of domain objects.

However, since both programming approaches emphasize behavioral extensions (i.e., explicit to formal design language transitions), there is still little support provided in the DODE model for tacit exploration. This lack of support for tacit exploration has a somewhat negative impact on design practices. In Chapters 2 and 4, we saw that an important practice of these designers is "designing their design representations." In this model, designers cannot engage in this

practice without system developer support; hence the new "designing representations" skill required by system developers.

Table 5.4: New Practices, skills and knowledge required by DODE Model.

Who?	New Practices	New Skills & Knowledge
Designers	• Customizing Tools	• end-user programming ?
Developers	• Locating Substrates • Evaluating Substrates • Designing Representations • Extending Substrates • Designing Environment Architectures • Understanding Work Practices • Linking Tools/Substrates	• expertise on substrate availability • substrate evaluation skills (extensibility, affordances, code stability) • participatory / collaborative design approaches • work practice analysis • domain analysis • general purpose programming languages • substrate-specific programming languages • tool linking protocols, integration services

While not many new skills are required by designers, the DODE model may require a new breed of system developer. Most of the new skills required by the model fall squarely in the realm of the developer. Many of these skills are probably new to the "average" system developer (e.g., work practice analysis, participatory design, domain analysis, substrate evaluation, tool linking and integration). This complex skill list raises the question of whether a single system developer can provide these services or whether a team of system developers is needed. While in principle a team approach is a viable solution, in practice, the design teams examined in this dissertation are quite small and there are important cost issues to consider. This issue of organizational support will be reconsidered in Chapter 7.

SUMMARY

This chapter presented and analyzed our experiences creating the Voice Dialog Design Environment. This domain-oriented design environment provided voice dialog designers with specialized design representations and knowledge-based mechanisms to support them in

constructing, simulating, and evaluating their designs. Overall, VDDE was successful at supporting design simulation and evaluation. However, it had surprising difficulties supporting design construction activities. In part, this was due to the rate of change in the domain; in part, it was due to the construction kit approach being rigidly applied to this dynamic domain. VDDE showed surprising flexibility, accommodating continual changes to domain vocabularies and some changes to the underlying representational system. However, considerable special skills and knowledge were required to make these changes. It is unlikely that designers would posses such skills and system developer support was required to make even minor changes to the system's design language. As our intuitions suspected, the presence of a rich domain model did hinder some aspects of the system's flexibility. In part, this inflexibility was due to the architecture and integration services adopted by the system. The findings in this chapter will be used in Chapter 6, when comparing DODEs with the Toolbelt model. Chapter 7 will use these analyses to inform the next generation of DODE design.

CHAPTER 6

COMPARING COMPUTATIONAL MODELS

In Chapter 4, three design environments based on the Toolbelt computational model were presented and analyzed. In this model, designers assemble and evolve collections of generic software applications to support making various design representations. The analysis revealed that our initial intuition concerning the flexibility of toolbelts was partially correct: the toolbelts examined seemed to favor some elaboration and modification processes. However, we also saw that the flexibility of toolbelts was limited at supporting other aspects of evolutionary change.

In Chapter 5, experiences from the Voice Dialog Design Environment project were analyzed. VDDE is based on the Domain-Oriented Design Environment (DODE) model. In this model, knowledge-based tools supporting the construction of customized design representations are created for specific design communities. Our experiences here also revealed that our initial intuitions concerning the support versus flexibility of DODEs were only partially correct. VDDE supported some aspects of use, such as design evaluation, quite well. However, we also saw that VDDE fell short as supporting other aspects, such as upstream design construction. Contrary to our intuitions, we saw that VDDE was surprisingly flexible at supporting some elaboration processes.

In this chapter, we'll compare the strengths and weaknesses of the general Toolbelt and DODE models using the domain construction framework outlined in Chapter 3. The purpose of this comparison is to better understand which aspects of these computational models help or

hinder particular domain construction processes. The chapter begins by comparing each models' support for domain construction processes. Next, key aspects of the underlying computational models – both technical and social – are compared and contrasted. Finally, other computational models besides these two are briefly considered.

DOMAIN CONSTRUCTION PROCESSES

This section compares how each model supports the domain construction processes involved in use, elaboration, and modification. Table 6.1 summarizes each model's support; the table entries for each model are based on the general reflections at the ends of Chapters 4 and 5. The purpose of these comparisons are to review the findings from previous chapters and to indicate which technical or social aspects of the underlying computational model appear to affect the process being considered. These aspects will be further discussed in the following sections.

Supporting Use

The two models support design languages at different ends of the tacit to explicit to formal spectrum. Toolbelts emphasize the tacit and explicit ends of the design language spectrum; i.e., most of the design language resides in practices and design products, not in formal domain models in tools. DODEs on the other hand, emphasize the formal end of the spectrum, providing tools or components with deep domain models. Construction components in particular, tend to provide little support for the tacit end of the spectrum. These differences appear to affect the models' respective abilities to support upstream design construction versus more downstream construction activities.

Table 6.1: Comparison of Toolbelt and DODE Models

Domain Construction Processes	Toolbelt Model	DODE Model
USE		
• full DL spectrum supported?	tacit / explicit only	explicit / formal only
• constructing	upstream favored	downstream favored
• evaluating - inspecting - analyzing - relating - envisioning & experiencing	Limited - visual inspection only - little support for relating	Yes, with limitations. - little support for relating except for heuristic types of relationships encoded in critiquing rules
• iterating	No	No
ELABORATION		
• DL transitions supported?	tacit to explicit only	explicit to formal only
• graphical refinement	Yes	Some. - *In situ* refinements not supported
• naming - simple data types (e.g., presentation-oriented name / look mappings)	Limited - No support in graphics tools	Limited. Requires Programming Knowledge
• object refinement - complex data types (e.g., semantic-oriented whole / part mappings)	Very Limited - In few tools only	Yes, with limitations - Additions and deletions problematic for design products - No instance refinements
• establishing relationships / design product integration	Possible, but difficult - Limited Direct Manipulation - High-level Event scripting - Low-level programming - Collection of ad hoc mechanisms - Mostly equality and transformation relationships	Possible, but difficult - No Direct Manipulation - Some form-based interfaces - Low-level programming usually required - Mostly heuristic relationships
• extension - object behaviors - tool extensions	Limited - tool extensions - Requires Programming Knowledge	Possible, but difficult - object behaviors - tool extensions - Requires Programming Knowledge
MODIFICATION		
• experimentation and creation	Requires Shopping and Exploration skills	Requires Programming Knowledge
• introduction - augmentation - replacement	Yes and No - Yes, if shallow model - No, if deep model	No
• retrofitting - no support - manual support - active support	Limited - Mostly manual - Active support limited to few tools only	No

Toolbelts provide little support for evaluation beyond visual inspection. In our design stories in Chapter 2, this lack of support had some negative effects on practices. The knowledge-based components in DODEs can provide support for design evaluation activities. However, this evaluation support also had some negative consequences for practices. The demands these knowledge-based components place on representational formality in the construction component can hinder the model's ability to support upstream design construction. In general, both models provide little support for maintaining relations across different design products, and as a result, neither support iterative design practices very well.

In summary, both models provide mixed support for use processes overall. Toolbelts favor use processes requiring informal design languages. DODEs favor use processes requiring more formal design languages. As yet, neither model appears to support design languages at mixed levels of formality, particularly in the area of design construction tools. This lack of support probably stems from affordances and hindrances of the underlying data models.

Supporting Elaboration

Again, the two models emphasize different ends of the design language spectrum. For the most part, toolbelts only support elaboration processes favoring tacit to explicit transitions. Most tools support *in situ* graphical refinement of design products. Some tools can be enriched with shallow domain models in the form of simple, presentation-oriented data types consisting of name/look mappings. Very few tools can be enriched with semantic-oriented domain models containing more complex data types (e.g., whole/part mappings). In Chapters 2 and 4, we saw that designers often took advantage of these semantic-enriching facilities where they were available. Even though most tools are extensible, it is difficult, if not impossible, to enrich these tools with deep domain models due to limitations of their underlying data model.

DODEs on the other hand, support elaboration processes favoring explicit to formal transitions. The underlying object-orientation of DODE data models favors semantic elaboration processes such as object refinement (whole/part mappings) and behavioral extensions (i.e., modifying existing and adding new attributes and methods to object class definitions). As we saw in VDDE, with proper structuring and substrate support, the domain model proved to be quite flexible in terms of supporting semantic refinements to domain vocabularies. However, it is unlikely that designers can perform these elaborations without system developer support due to the special skills and knowledge required. In the future, new end-user programming languages such as Visual AgenTalk [97] may enable designers to perform some extensions on their own; this possibility will be further considered in Chapter 7. However, in the DODEs constructed to date, some of the rigidities of the underlying object model have been directly mirrored in the interface of the construction component. These rigidities make it difficult for DODEs to support more tacit, presentation-level elaboration processes such as graphical refinement and naming in the construction component.

Toolbelts, in theory, support some forms of integration such as the establishment of equality and transformation relationships. Some tools enable equality relationships to be established using direct manipulation mechanisms similar to existing copy/paste interfaces. Some tools support across-application high-level, event scripting or task automation languages capable of representing many transformation relationships. Many recent application linking protocols (e.g., OpenDoc or OLE) could support more integration relationships. However, these protocols are programmatic integration mechanisms intended for system developers and not end users. In general, support for any of these integration mechanisms is sporadic and unpredictable; it appears to be a rare occurrence for all tools in any given toolbelt to support the same integration mechanisms.

Most DODEs constructed to date have focused on supporting heuristic relationships only. For the most part, only programmatic

integration interfaces are provided and therefore designers need system developer support in establishing relationships between design products. A few DODEs have explored end-user mechanisms for establishing these relationships with limited success [38, 76]. Recently, DODEs have been created supporting the establishment of other forms of relationships using form-based interfaces. EVA [84] supports establishing correspondence relationships between interface and code objects by filling in special forms associated with interface objects. X-Network [106] provides a general-agent mechanism that can be used to create various types of relationships. A special form-based agent editor is provided for defining an agent by selecting from provided lists of attributes and actions. However, these two promising approaches are still primarily targeted at supporting system developers, rather than designers, to establish relationships.

In general, both models provide mixed support for elaboration processes. Toolbelts are less flexible than anticipated in the sense that they cannot be enriched with deep domain models at all. DODEs were more flexible than anticipated in the sense that they accommodate a wide range of elaboration processes, though taking advantage of these processes requires system developer support. The differences in the types of elaborations supported appears to stem from differences in the underlying data models. In general, integration services in both models only provide designers with limited support for establishing relationships between design products. Both models provide an overly diverse collection of integration mechanisms that must be individually learned. It is rare for a single tool to support multiple forms of relationships and it is rare for these mechanisms to be usable by designers themselves. The area of integration services represents perhaps the single biggest hole in both models with respect to supporting domain construction processes.

Supporting Modification

The two models are both similar and different in their support for modification processes. Toolbelts, with their support for informal design languages, enable designers to experiment with and create new

representational systems on their own. DODEs, with their emphasis on formal design languages, require designers to have developer support for these processes.

As long as the tools in Toolbelts are not enriched with deep domain models and relationships across tools, then introduction processes are straightforward. However, as soon as tools are enriched in these ways, Toolbelt support for introduction processes begins to resemble that provided by the DODE model; i.e., introduction of new representational systems becomes costly. This similarity stems from both models being based on coalition-style architectures relying on point-to-point integration services and completely distributed domain models.

Neither model supports retrofitting activities very well; i.e., changing old design products to conform to new design languages. There are basically three levels of support. Level one provides no support at all; i.e., old design products are probably not even readable by the design environment. Level two at least enables designers to manually retrofit by allowing old design products to still be opened in the evolved design environment. At level three, design environments actively support designers in the retrofitting process by assisting them to resolve differences between design language versions. In general, Toolbelts provide manual (level 2) support; a few tools provide limited forms of active (level 3) support. In DODEs, even the ability to manually retrofit can be hindered if the underlying object model cannot restore a design product based on an old design language version.

TECHNICAL ASPECTS OF THE COMPUTATIONAL MODEL

The previous section identified several technical areas of computational models that affect domain construction processes: data models, architectures, domain models and integration services. In this section, we will compare and contrast the Toolbelt and DODE models in these areas in order to better understand the benefits and limitations of

the different approaches and to identify where more work needs to be done.

Data Models

In previous analyses, it emerged that two types of flexibility should be supported by the underlying data model:

- *in situ* refinement of design vocabularies and
- mixed levels of design language formality.

In situ refinement refers to the ability to refine individual instances of domain vocabulary items directly within the context of a specific design product. Mixed levels of formality refers to how tools should support the full spectrum of elaboration processes and make the results of these elaborations available to designers for reuse. The underlying data model used in the design environment appears to impact these two types of flexibility. This section will compare three different data models – procedural-based, object-oriented class inheritance, and object-oriented prototype inheritance – with respect to their support for these two types of flexibility (Table 6.2). The tools in toolbelts have procedural data models while DODEs have object-oriented data models.

Table 6.2: Comparison of three data models.

	Procedural	**Class Inheritance**	**Prototype Inheritance**
in situ refinements	Yes	No	Yes
mixed levels of formality	Informal	Formal	Mixed

Procedural Data Models

For the most part, the tools in toolbelts support *in situ* refinement, but only support design languages in the tacit to explicit end of the spectrum. Even though most tools in toolbelts are extensible, it is difficult, if not impossible, to enrich these tools with formal domain models. This is because most tools do not support the concept of domain "objects", at even the superficial naming level. It is only

recently that extension languages in these tools have looked beyond simply providing procedural hooks and into exposing the underlying data model also. (See [87, 121, 122] for more details.) Both access to the underlying data model and extensibility of the data model are crucial for supporting deeper domain models. This emphasis on procedural versus data model extensibility is probably historical, stemming from the fact that most of these tools are implemented (or used to be implemented) in procedural rather than object-oriented programming languages. However, as we shall see in the following discussion, this lack of object-orientation probably contributed to the tools' overall flexibility, particularly in the area of supporting *in situ* refinements.

Class Inheritance Data Models

Many DODEs constructed to date have used class inheritance object models [113]. For the most part, the object models have been mirrored directly in the interface of the construction component. Specifically, each domain vocabulary item in the construction kit has a corresponding class definition in the object hierarchy. To introduce a new vocabulary item requires introducing a new class definition. Modifying a single vocabulary item requires modifying the entire class definition, and thus the change is applied to all instances uniformly. In effect, elaborating the construction component requires some awareness of the underlying object hierarchy.

Mirroring the object model in the construction component's interface contributes towards the increased level of formality required to make any modifications and makes it difficult to support less formal elaborations such as graphical refinement and naming. The names of vocabulary items provide the mappings between the construction kit interface and the underlying object model. Thus, they are the province of the object model, not the designer; changes cannot be made without severing the link or affecting the object hierarchy. The strict mirroring of the class inheritance mechanism in the interface inhibits supporting *in situ* graphical refinements of instances; refinements can be performed at the class level only.

Prototype Inheritance Data Models

The procedural orientation of toolbelts appear to favor informal design languages and tacit elaboration processes, but also appear to hinder the elaboration of more formal domain models. The class inheritance orientation of some DODEs favors formal design languages but appears to lack the flexibility required to support more tacit, instance-level elaboration processes. Some DODEs have been constructed using an alternative object model based on prototype inheritance mechanisms [66]. According to Shipman [106], the benefit of this approach is that it removes the system-oriented class/instance distinction from the interface which makes it more suitable for supporting mixed levels of formality and *in situ* refinements.

Shipman constructed a special prototype-based hyper-object substrate (HOS) and studied its use in several graduate class projects. Initial experiences using HOS indicated that prototype inheritance did support many forms of *in situ* elaborations. However, the particular instantiation in HOS had some shortcomings with respect to supporting radical modification activities involving the extensive modification and removal of domain information. Also, this approach was applied mainly to textual information. The one DODE created (X-Network) assumed a static representational system (a logical map) in the construction component, thus it remains to be seen how well this specific object model substrate could support the evolution of graphic design language elements.

In general, the prototype inheritance object model looks like a promising approach for design environments constructed for dynamic and innovative domains. More work needs to be done to investigate: (1) how such mechanisms can better support radical modifications and removal of existing information, and (2) how to apply the model to graphic information in the construction component to support elaborations and modifications to representational systems.

Architectures

In previous analyses, several issues surrounding architectures emerged; specifically, the notion that domain models and integration services affected the overall flexibility of the environment. In this subsection, we will consider how these issues affect the type of evolution supported by a particular architecture by examining the effects of:

- distributed versus shared domain models, and
- point-to-point versus data-oriented integration services.

In this dissertation, a domain model includes computational representations of domain vocabularies, representational systems, and relationships between these two. This domain model can be shared by all tools or components in a design environment or distributed across them. Integration services refers to the mechanisms available for integrating or linking domain information across components; i.e., the mechanisms for representing the relationships between domain objects. Two types of mechanisms are considered: point-to-point mechanisms where components are integrated with each other or data-oriented mechanisms where components are integrated with underlying shared domain models. These choices appear to impact the type of evolutionary processes favored by particular architectural styles. This section will compare these features across three different architectural styles – coalition, shared object model, and extended substrate – in order to better understand how these choices affect flexibility. Table 6.3 summarizes the results of this analysis.

Table 6.3: Comparison of three architectural styles.

	Shared Object Model	**Coalition**	**Extended Substrate**
Domain Model	Shared	Distributed	Mixed
Integration Services	Data-oriented	Point-to-Point	Mixed
Type of Evolution Supported	Elaboration (Vertical)	Modification (Horizontal)	Both ??

Figure 6.1 shows two different architectures examined in this dissertation. Coalition style architectures are used by Toolbelts and by some earlier DODEs. Shared Object Model architectures are used by other later DODEs. Feiler and Wallnau examined these two architectural styles in the historical context of CASE tools [24]. According to their analysis, the two architectural styles support different types of evolution.

Shared Object Model Architectures

Environments based on shared object model architectures support vertical evolutionary processes (Figure 6.1, left side). In CASE tools, environments evolved upward as shared object systems evolved into shared data models and then tools were created using the shared data models. In DODEs based on this architecture (i.e., X-Network, EVA, Hermes), we also observed downward evolution in the sense that designers and system developers were able to further refine shared object models using the provided tools and integration service mechanisms. According to Feiler and Wallnau, CASE tools based on this architecture tend to offer better integration between tools. In this architecture, individual tools integrate with the infrastructure and integration across tools is achieved by data sharing. When one tool changes the data model, the change is visible to all tools. Thus, this data-centric architectural style seems to favor supporting elaboration processes involving enriching tools with domain objects and domain relationships.

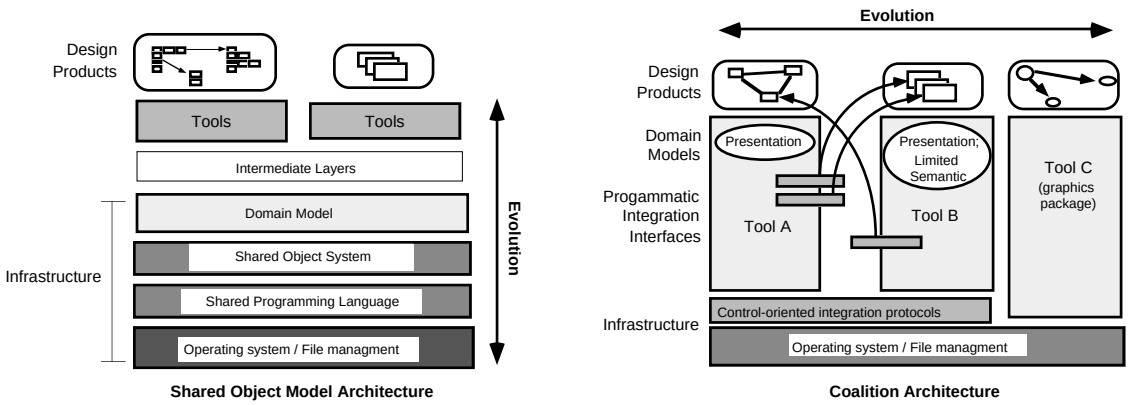

Figure 6.1: Shared Object Model and Coalition Architectures.

However, as Feiler and Wallnau noted, this architectural style does not favor horizontal evolution in the sense of supporting new tools or components being added as needed. For one thing, this architectural style requires consensus across components about what should be in the shared object model. New components have to be constructed to conform to the shared data model and existing components cannot modify the shared model without considering the possible consequences to other components. Thus, this model introduces many component interdependencies that make it difficult to carry out radical design language modifications.

Coalition Architectures

Most DODEs, Toolbelts, and many CASE environments are based on coalition architectures which seemingly offer greater flexibility in the area of supporting radical modifications. Coalition architectures support horizontal evolutionary processes by enabling new tools to be more easily introduced into the environment (Figure 6.1, right side). This tool-centric architecture requires no coordination and consensus across tool developers. Tools can be developed independently, by different developers at different periods of time, and can be introduced into architectures as required. In the VDDE project, the critiquing subsystem was independently developed and introduced into the system two years after the initial creation of the VDDE construction/simulation area. Thus, coalition architectures seem to favor modification processes such as creating and introducing new tools and representational systems into design environments.

However, as found in CASE tools, Toolbelts, and DODEs, environments based on this architecture tend to offer ad hoc collections of integration services and uneven support for elaboration processes. Additionally, when individual tools and environments are enriched with deep domain models (i.e., lots of domain objects and domain relationships), the resulting environment loses the flexibility it had. The use of point-to-point integration mechanisms makes it difficult to replace existing tools. The completely distributed domain model makes it costly to introduce new tools as they must be enriched

completely from scratch since there is no data sharing in this architecture.

Extended Substrate Architectures

Neither of these two architectural styles, in their extremes, are satisfactory for supporting both elaboration and modification processes. Different forms of architectures merging aspects of these two styles are required to support both modification (horizontal) and elaboration (vertical) forms of evolution. New architectural trends are already happening to some degree. In Toolbelts, DODEs, and CASE the trends are similar in that new intermediate layers are being introduced. However, in these three kinds of models, different types of intermediate layers are being investigated.

Currently in the Toolbelt model there is a shift from tool-centered integration services to document-centered integration services [87]. This shift is an attempt to deal with compound documents containing a variety of media types in a more seamless fashion. As a result of these efforts, several competing compound document architecture standards are emerging [87, 121, 122]. In general, these architectures specify document layout frameworks that indicate how different parts can be accommodated in the same document, automation mechanisms for cross-application scripting, and protocols guiding how applications will exchange document parts and objects. As such, these services represent a change in architecture for the Toolbelt model by introducing a new layer of cross-tool, shared infrastructure support.

Recent DODEs, such as ProNet and VDDE, introduced tool substrates as intermediate layers in the architecture. As discussed in Chapter 5, substrates enhance the environment's overall flexibility for supporting elaboration processes. However, simply introducing substrates into an otherwise pure-coalition architecture does not help with modification processes when deep domain models are present. Changes to integration services are required that help to reduce the reliance on point-to-point integration mechanisms. Perhaps trends occurring in the Toolbelt model could be combined with substrates in the DODE

model to produce a new architectural style. Figure 6.2 offers one speculative view of what such an extended version of a substrate architecture might look like. Basically, this extended architecture postulates that new layers need to be introduced into the infrastructure that provide for some partial sharing of the domain model and thus help reduce the reliance on point-to-point integration mechanisms. This layer might be implemented as a shared object model layer or as an object exchange protocol similar to the trend in the Toolbelt model. Either way, introducing such a layer requires reconceptualizing aspects of substrates; specifically, in the areas of object management facilities and domain models.

In summary, this analysis showed that architecture is a major determinant of an environment's overall ability to support elaboration and modification processes. Certain architectural models favor some processes at the expense of others. This analysis in particular highlighted the impact of domain model distribution and integration services on an architecture's ability to support evolution. More research is needed into architectural styles capable of supporting both kinds of environment evolution. Emerging architectural trends favor the introduction of new intermediate layers providing shared infrastructure services, although the exact nature of these new layers is still being investigated.

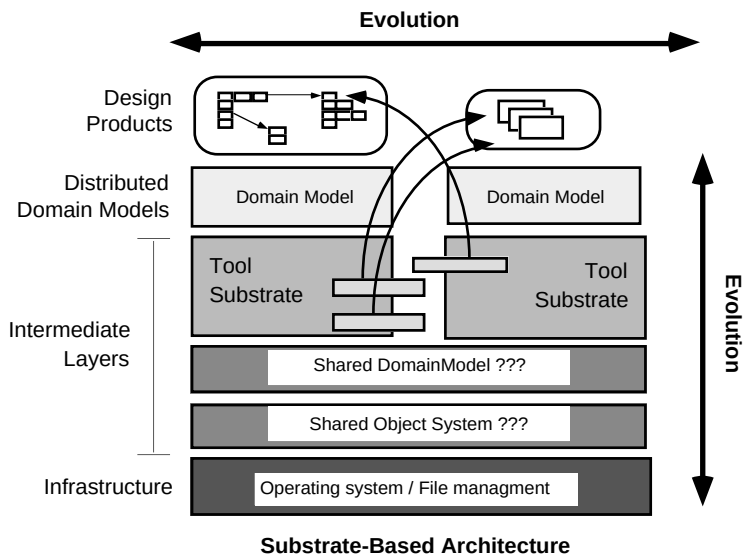

Figure 6.2: Extending the substrate-based architecture.

SOCIAL ASPECTS OF THE COMPUTATIONAL MODEL

Chapter 3 outlines a socio-technical definition of a computational model. In the previous section, we considered technical aspects of computational models. In this section, we'll consider the social aspects of computational models. Particularly, we'll compare the new practices, skills, and knowledge required by Toolbelt and DODE models and discuss the implications of these new requirements for environment ownership and organizational support.

New Practices, Skills and Knowledge Required

Table 6.4 enumerates the new practices required by the two models as derived in Chapters 4 and 5. New practices fall into two broad categories: those that can be assumed by the designers themselves and those that are probably assumed by system developers.

Table 6.4: New Practices required by Toolbelt and DODE Models.

Who?	Toolbelts	DODEs
Designers	• Locating Tools / Shopping (*) • Experimenting with Tools (*) • Assessing Tools (*) • Customizing Tools (*)	• Customizing Tools
Developers	• Extending Tools • Linking Tools	• Locating Substrates • Evaluating Substrates • Extending Substrates • Designing Environment Architectures • Linking Tools/Substrates • Designing Representations • Designing Work Practices

Toolbelts introduce many new tool-oriented practices into design activities. In Chapter 4, we saw that designers assumed many of the practices themselves. We also saw that designers had difficulties with some of these new practices and could have benefited from organizational support in these areas (*). In our stories, many tools

were never significantly extended or integrated. For the most part, this was because tools did not provide compatible extension or integration mechanisms. However, even if tools were extensible in these areas, the low-level programming required would probably render such extensions outside the scope of most designers' knowledge and interests. Given current mechanisms, these extensions are probably best performed with system developer support.

DODEs also introduce many new practices into the workplace. Most of these new practices are tool-oriented; a few are work practice-oriented (e.g., design representations and work practices). In Chapter 5, we saw that system developers were needed to perform many of these new practices. Making any changes to tools and domain models, even simple ones, at least requires technical knowledge and oftentimes requires actual programming effort. Due to the technical knowledge required, designers cannot design the necessary representations and related work practices themselves, but instead must rely on collaborations with system developers to assist them in these activities.

Organizational Support Requirements

Designers could benefit from organizational support in both Toolbelt and DODE models; however, the two models have quite different organizational support *requirements*. Here we will briefly consider these support requirements and how they vary throughout the lifecycle of a design environment. Specifically, we will examine the system developer assistance required during the following lifecycle processes:

- creating an initial design environment,
- design language elaboration, and
- design language modification.

In the Toolbelt Model, creating an initial design environment primarily involves assembling tools. Modifying a design language often involves replacing old tools with new ones. Thus, the practices pertinent to creating an initial design environment and modifying the

design language are very similar: namely, locating tools, experimenting with tools, and assessing tool suitability. As shown by Table 6.1, while designers could benefit from organizational support for these activities, such support is not required. In the three design stories, designers had taken on these practices themselves. With respect to elaboration processes, as long as designers are using tools in a generic way or only enriching tools with shallow domain models, little or no system developer assistance is required. However, extending tools with deeper domain models or integrating tools would require developer support.

As implied by Table 6.1, DODEs in their current form, would probably require organizational support on a fairly continual basis during the entire environment lifecycle. When creating an initial design environment, a team of developers could be required, from the standpoint of both resources required and skill coverage. Once the system is deployed, as we saw in the VDDE project, developer assistance is still required to support most elaboration and modification processes.

This prediction of continued developer support differs somewhat from the proposed DODE vision. The DODE model advocates a design environment lifecycle based on the seeding, evolutionary growth, and reseeding process model[32]. During the seeding phase, system developers work with designers to create an initial design environment. In the evolutionary growth phase, designers add information to the seed as they use it to create design artifacts. Reseeding occurs when system developers assist designers to reorganize and formalize information previously added. Thus, in the SER model, developer assistance is not required during the evolutionary growth phase.

There are at least two explanations for these different predictions concerning developer support. First, the assumed rate of change in the SER model appears to be slower than that observed in innovative and dynamic domains. Thus, since changes occur more frequently and these changes require developer assistance to enact, then more overall developer support is required. Second, efforts to facilitate evolutionary

growth have focused on supporting incremental formalization of information in knowledge-bases such as design rationale. However, as we saw in the design stories, domain vocabularies and representational systems also need to undergo continual change in dynamic domains. Thus, until DODEs provide mechanisms that support designers to perform these types of elaborations themselves, more continuous system developer support will probably be required.

Environment Ownership Considerations

Throughout most of this dissertation, we have considered design environments from a functionality perspective, examining whether the proposed environment can provide the necessary support and flexibility. In this section, we consider how the proposed environment affects the organizational empowerment of the designers it is intended to serve.

The previous analyses of new practices and organizational support requirements show that the toolbelt model requires more, tool-oriented, potentially burdensome practices to be assumed by designers than the DODE model. However, the same analyses show that the DODE model requires designers to share work practice design activities, that previously were under their control, with system developers, and to some extent to be dependent upon these system developers. While such collaborative system development requirements may be politically empowering for some end-users [84], for the designers considered in this dissertation, it reflects a loss of control and ownership.

In [119], we traced the historical motivations behind the toolbelt model and concluded that part of its success was due to its being a very politically empowering model. In the early days of system design (1970's), MIS departments were powerful, centralized bodies controlling the software development process, often at the expense of workers. Whereas the participatory design movement arose in Europe to deal with the inequities prevalent in this situation, other factors contributed to an entirely different antidote in the US: the rise of end-user com-

puting (EUC) [11]. Personal computers and generic off-the-shelf software proliferated as workers were able to use their local budgets to select and purchase the computational tools of their choice. While the low-cost economics of personal or end-user computing undoubtedly contributed to its popularity, many claim that the desire of users to escape from MIS dominance and have more control over their tools also contributed to its rapid success [19].

Now, in the mid-1990's, many industries are faced with increasing global competition, rapidly changing markets, and decreasing budgets. This situation is giving rise to flatter organizational structures where small, self-managed, independent teams are increasingly responsible for everything [85]. These teams are particularly prevalent in dynamic and innovative domains such as software design. In these domains, it is increasingly part of the new "job requirement" for designers to design how they will do their job and what computational tools they will use.

Toolbelts are part of this politically empowering end-user computing phenomena. The design communities considered in this dissertation are examples of politically empowered, small, independent teams. To some extent, domain construction is what happens when these two trends are rubbed together; that is, when empowered designers are given the means and the opportunity to evolve design languages to better support changing work practices. Against this backdrop, DODEs represent a loss in ownership and control in the sense that power over the design of local work practices is being returned to the system developers.

Thus, ownership is an important issue to consider when introducing domain-oriented tools in to the workplace. The key question is who is in control of the "domain-orienting" process? In Toolbelts, designers are in control of the domain-orienting process. In DODEs, this process is shared between designers and system developers. If the environment being introduced takes control away from designers, then there could be resistance to its adoption no matter what

functionality the system provides. Also, this analysis re-emphasizes the importance of examining power relations in a specific context; i.e., what is gained and what is lost by introducing a specific environment can only be gauged by comparing it to the current workplace setting.

OTHER COMPUTATIONAL MODELS

This dissertation has focused on comparing design environments based on two specific computational models. In this section, we will briefly consider other computational models being investigated from the domain construction perspective.

Programmable Design Environments

Programmable Design Environments (PDEs) have been created in graphic domains such as chart making [22] and paper sculpture [23]. PDEs combine direct manipulation interfaces with general-purpose programming languages that have been enriched with domain-oriented constructs [21]. As such, the environments are very extensible by people with considerable programming skills. Some PDEs provide mechanisms, such as self disclosure [17], to assist designers in acquiring the necessary programming skills. However, while PDEs are extensible, they do not directly support the design language evolution processes of elaboration and modification. Particularly, PDEs do not support incremental formalization since changes to the environment must be formally expressed in programming code.

Basically, PDEs are based on a philosophy that may not be congruent with dynamic and innovative domains. In our design stories, we saw that design tools and design representations were somewhat short-term and disposable in the sense that when they significantly broke down, designers were very willing to replace them. The PDE approach assumes that designers are willing to invest significant effort to learn the programming language and to enrich the system with domain-oriented constructs in the interest of creating an environment to support them over the long term. As such, PDEs may be better suited

for more stable domains or as intermediate substrates in a design environment architecture.

The Application Construction Environment

The Application Construction Environment (ACE) [59] has been used to create interactive information-intensive applications, such as stock analysis applications, where users need to view and analyze potentially large amounts of data. ACE is a very promising environment that seemingly supports many domain construction processes.

ACE consists of a layered architecture with three layers corresponding to different developer/user roles. At the lowest layer is ACEKit. ACEKit is a set of C++ libraries providing infrastructure support services such as saving and restoring objects, object communication protocols, and object change notification broadcast mechanisms. In the middle layer, ACE provides libraries of visual formalisms [79] and user interface selector objects [58]. System developers combine and specialize visual formalism and selector classes and add application-specific data types to create individual applications. At the top layer, ACE provides an end-user programming language based on spreadsheet formula languages. This language enables end-users to extend the functionality of visual formalisms, to operate on the content of visual formalisms, and to add new application data types.

The support for incremental refinement, the end-user programming language, and the infrastructure support for integrating across visual formalisms suggests that ACE supports many of the proposed elaboration processes. The ability to introduce new visual formalisms into an environment suggests that ACE provides some support for modification processes. However, ACE was primarily designed to meet the rapid application development needs of C++ programmers [128]. Some extensions, such as adding new application data types or visual formalisms, require the application to be re-linked;

thus the extension mechanisms are not as smoothly integrated into the environment as desirable from the domain construction perspective.

To date, ACE has been used primarily by application developers and there have been few published reports of end-user experiences. Thus, it remains to be seen how flexible the particular mechanisms provided by ACE are and whether end-users (designers) can perform the necessary extensions themselves. However, ACE does provide a good example of an architectural approach to environment design; features deemed desirable at the end-user level influenced choices and design decisions made at lower, infrastructure levels. Its use of visual formalisms offers an alternative perspective on intermediate layers that is similar to the use of substrates in DODEs.

SUMMARY

Toolbelt and DODE models emphasize different ends of the tacit to explicit to formal spectrum. These differences affect their respective abilities to support upstream design construction versus more downstream construction and evaluation activities.

Underlying data models were shown to affect an environment's ability to support design language elaboration processes. In Toolbelts, lack of access to underlying data models inhibits enriching tools with deep domain models. In DODEs, much of the apparent inflexibility stems from limitations of the underlying object-oriented class inheritance formalism that are manifested at the interface level. Prototype inheritance mechanisms may be more suitable for supporting design language elaboration processes.

Software architectures were shown to be an important factor affecting an environment's ability to support both elaboration and modification processes. Issues affecting an architecture's flexibility include domain model distribution and the type of integration service provided. The point-to-point integration mechanisms used by

Toolbelts and DODEs hinder both models abilities to support modification processes.

Both models introduce new tool-oriented practices into the work place. In the Toolbelt model, many of these new practices can be assumed by the designers themselves. In the DODE model, system developers are required to perform the new tool-oriented practices and to participate in the design of work practices. An analysis of changes in control (of work practice design) and environment ownership argued that, in its current form, the DODE model resulted in a loss of empowerment for the types of designers being considered in this dissertation.

CHAPTER 7

NEXT STEPS

In previous chapters, we analyzed and compared two computational models against our empirically-inspired domain construction framework. We saw that toolbelts had many desirable qualities but suffered some negative consequences due to their generic nature. We also saw that the domain-orientation of DODEs provided several benefits but was perhaps too strict in some respects. The empirical studies and the previous analyses indicate that the question of whether generic or domain-specific software is better suited for some areas of design practice may be too simplistic. In fact, *what these designers need are tools that bridge the gap between these two extremes by combining elements of both.*

In the following sections, we will discuss how the Toolbelt and DODEs model can be extended to include some beneficial elements of each other. The comparison of technical and social aspects of computational models in Chapter 6 pointed towards possible areas where improvements could be made or more research is required. This chapter will examine these possible areas in order to make recommendations for the next generation of design environments based on each model.

This chapter begins by discussing how the Toolbelt Model could be improved through tool re-design and new forms of organizational support. Next, the DODE model is considered and several areas requiring further research are enumerated. Finally, the chapter concludes by speculating on what it means to "bridge the gap" and whether this ideal requires entirely new computational models.

IMPROVING TOOLBELTS

As the design stories in Chapters 2 and 4 show, designers often use the generic applications in toolbelts in very domain-specific ways. Over time, design communities create graphic vocabularies for expressing important domain concepts and well-defined representations for making important concepts and relationships visible. The flexibility and formatting features of generic tools enable designers to continually evolve their vocabularies and representations to better support changing work practices. However, the tools' generic nature has several negative side effects such as introducing cognitive and manual burdens on constructing and maintaining designs and hindering iterative design. These problems could be mitigated if designers were able to enrich tools with some awareness of their design languages; i.e., if these otherwise generic tools could be made a little more domain-specific.

However, designers encounter numerous problems when trying to evolve their toolbelts in this direction. First, many tools do not support the necessary design language elaboration processes. Second, designers may lack some of the necessary skills and knowledge to evolve their toolbelts. Thus, improvements to support toolbelt evolution fall into two broad categories:

- re-designing tools and
- improving the use of toolbelts through organizational support.

The following speculations about how to improve future toolbelts offer both specific recommendations and general pointers to where additional work is needed in these two categories.

Re-designing Tools

Tool tailoring facilities particularly need to be improved to support design language elaboration processes in two, specific areas: support for

enriching tools with domain vocabularies and support for establishing relationships across tools. Each of these areas will be considered in turn.

A key difference between generic and domain-oriented tools is that generic tools lack domain-specific data types (i.e., domain vocabularies). However, many tailoring facilities in generic tools are oriented towards task automation and extending tool functionality. Very few tools support designers to enrich tools with awareness of domain vocabularies; where tools offer this functionality, we observed that designers often took advantage of it. Thus, one recommendation is that tailoring facilities should enable designers to articulate and refine domain vocabularies. As enumerated in Chapter 3, this requires tools to support elaboration processes such as:

- defining simple data types (e.g., specifying and modifying name/look mappings),
- defining more complex data types with multiple, named components (e.g., specifying and modifying whole/part mappings),
- attaching behaviors and procedural actions to vocabulary items or parts of vocabulary items, and
- making these user-defined vocabulary items available for reuse.

As discussed in Chapter 6, a big deficiency in the toolbelt model is lack of support for establishing relationships; i.e., integrating individual tools to form a coherent "system." While some integration is theoretically possible, in practice most tools provide little or no support for integration. Where possible, it often requires extensive low-level programming. This is a challenging problem from both the integration services perspective and the interface perspective. In the previous chapter, new developments at the integration services level were described; thus there is hope that integration across tools will be possible in the near future.

However, these developments are targeted at application developers, not end-users. And, as we saw in our design stories, it is crucial that end-users be able to establish and modify relationships themselves. While there is much research interest in end-user programming and tailoring mechanisms, relatively little attention has been paid to this specific area of cross-application tailoring by end-users. More research is needed in this relatively new area of tailoring. If the current industry trend towards componentware [122] continues and surfaces at the customer level, then these interoperability interface issues will become increasingly important.

This dissertation also illustrates how studies of work practices can provide insights into these issues. Specifically, we observed designers acting as "human glue" binding applications together. Analyses of these situations showed that: 1) relationships across applications fell into four broad categories, 2) relationships were fine-grained in the sense that there could be complex relationships between many different object parts across tools, and 3) many relationships were based on the look and content of *domain-specific* objects. These findings should be the starting point when investigating future interfaces for interoperability lieing between direct manipulation and general purpose programming (Figure 7.1).

Organizational Support

In our design stories, we saw that designers did experience some difficulties evolving their toolbelts and could have benefited from organizational support. The problems observed fell into two categories: locating tools and extending tools.

Specifically, designers had difficulties locating tools in the diverse and ever-changing marketplace and they had difficulties assessing the suitability of tools. Sometimes, designers were not sure if the desired tools did not exist or if they were simply not able to find them. While it is already difficult to assess high-functionality tools with hundreds of features, it becomes even harder when issues such as extensibility and interoperability also must be considered.

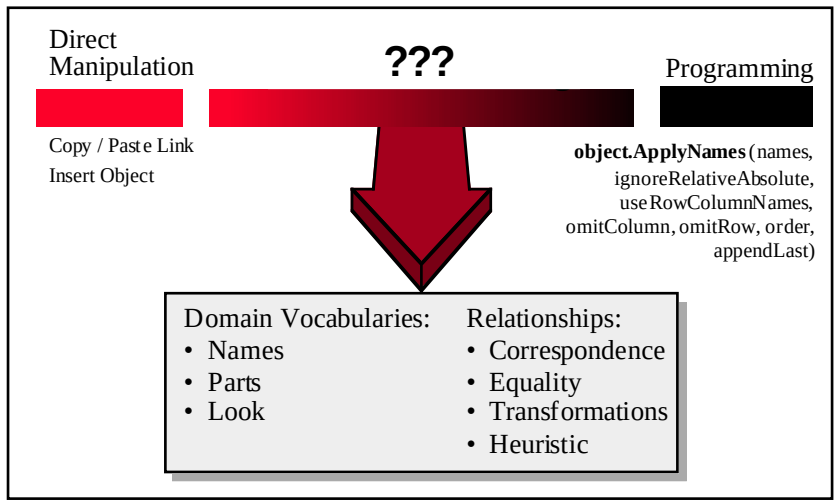

Figure 7.1: Designing interfaces for interoperability.

Most tools offer interfaces at the extreme ends of the spectrum: end-users can create relationships using direct manipulation or system developers can create relationships using integration protocols. The middle ground between the two endpoint should be explored, particularly expressing relationships in terms of domain-specific vocabularies.

Organizations should provide small design groups with support for locating and assessing tools. It is a full-time job keeping up with developments in the software marketplace; it is unreasonable to expect that designers will have the time, interest, and knowledge to fulfill this role when they have other, primary duties to attend to. Such a role could be fulfilled by "tool centers" containing people that have developed expertise in a few specific tools, broad knowledge of a variety of software tools, tool exploration skills, and awareness of market trends. Designers and other tool users could use the tool experts on an as-needed consultant basis. Difficult issues to be resolved include how centralized or distributed should these tool centers be; i.e., one per organization or multiple centers dispersed throughout the organization? In general, this centralization issue is not new in organizational computing management discussions and is not easily resolved.

It also should be noted that the notion of tool centers does run counter to many concerns in information system management. Current debates in managing end-user computing resources fall into two camps: those advocating control of end-user computing (EUC) resources and those advocating support [11, 73]. In general, advocates of control want to limit discretionary purchasing and tool selection. However, as shown in our design stories, empowering designers to design their own work practices and tools can yield powerful results; thus policies emphasizing support seem preferable, in these cases, to policies emphasizing control.

As previously discussed, designers could benefit from extending tools with deeper domain models and integrating tools to better work together. However, most tools providing such extension and integration facilities currently offer only low-level programming facilities beyond the scope and interest of many designers. Thus, taking advantage of these facilities would require developer support. Such needs could probably be met by a variety of part-time personnel support forms, each with its own strengths and weaknesses.

One approach is to encourage individual domain professionals to acquire extended computer knowledge and have these "translators" [67] or "local developers" [36], support other co-workers in their adaptation needs. This approach has the benefit of providing part time support resources with local knowledge. However, local developers often lack the detailed technical knowledge necessary for making more robust and in-depth tool adaptations. Therefore, the danger with this approach is that toolbelt evolution becomes a shortsighted, tinkering process [119, 120].

Another possibility is to have professional system developers support the process of toolbelt evolution. Developers possess the in-depth technical knowledge necessary for adapting and evolving tools and could be available to workgroups in the form of intermittent, roving personnel resources. However, the developers lack of local knowledge will make it difficult for them to support practitioners in consciously

evolving their work practices or their design representations. Thus, the danger with this approach is that toolbelt evolution will become highly technology driven, ignoring important opportunities for improving work practices [5].

Some people are concerned that supporting groups to evolve their tools will lead to diverging group work practices or even "social demassification" [12]. In many groups and organizations, smooth functioning depends on shared conventions, tools, and practices. A primary concern is that practitioners will tailor their tools in radically individualized ways and this will lead to diverging, and even chaotic, work practices and working environments. These are valid concerns that, again, will not be fully understood until we have observed Toolbelt evolution over a long period of time. However, I believe that this is an unlikely outcome; that instead, tool tailorability promotes convergence in work practices. For instance, Trigg and Bodker studied a work group tailoring Word Perfect to support their work practices [120]. They found that over time, systematization emerged as the group took advantage of the tool's tailorability to co-evolve their tools and work practices. As another example, the three design communities we observed were using generic off-the-shelf applications which offered hundreds of features and many forms of customization. These communities had plenty of opportunities to diverge. However, they did not. Instead *we saw convergence*, as they enriched their tools and evolved practices to promote the production of shared, standardized design representations.

IMPROVING DOMAIN-ORIENTED DESIGN ENVIRONMENTS

In the VDDE project in Chapter 5, we saw that domain-oriented design environments can provide support for design evaluation and design simulation activities. We also saw that, with system developer support and an underlying substrate, such environments were quite flexible in terms of supporting design elaboration processes concerned with creating deep domain models. However, we also observed problems in two general areas. First, there were unexpected

breakdowns in supporting use and elaboration processes concerned with shallow domain models that stemmed from the environment's overall lack of support for informal design languages. These problems could be mitigated if domain-oriented systems could more flexibly accommodate informal design vocabularies; i.e., if these otherwise domain-specific tools could accommodate less-formal, generic elements also. Second, we also saw that some technical aspects of the system's underlying architecture had adverse affects on some elaboration and modification processes. Thus, suggested improvements to DODEs fall into three broad categories:

- improving support for informal design languages by re-conceptualizing some DODE components,
- improving support for elaboration processes with alternative object models, and
- improving support for modification processes by considering underlying architectural issues.

The following discussion about how to improve future DODEs in these areas is admittedly speculative. However, analyses in previous chapters do indicate both promising directions for future research and areas where more work is needed. The purpose of this section is not so much to make specific recommendations but, instead, to enumerate key areas where future work can contribute towards moving the DODE model from being primarily a research model to being a viable workplace model.

Reconsidering Architectures

In chapter 5, we saw in the architecture discussion that there were few invariants when considering the general DODE architecture. This is to be expected in a research model where many issues are still at the exploratory development stage. One contribution of this dissertation was to compare these existing architectures in a systematic way and to relate them to previously studied architectures in related fields. Based on these analyses, we saw that the two existing architecture models – component-based (coalition) and shared object-based – favored

different types of evolutionary processes. We also saw that when deep domain models were created with extensive cross-component relationships, the component-based model lost much of its flexibility. Two problems with the component model were its exclusive reliance on point-to-point integration mechanisms and a completely distributed object model. Thus, future research at the architectural level should focus on ways to reduce the reliance on point-to-point integration and also consider mechanisms for sharing parts of the domain model across component boundaries. Chapters 5 and 6 discuss these issues in more detail. Overall, supporting modification processes in the presence of a deep domain model is an important issue to be resolved for the DODE model to be viable in dynamic domains. People will not be willing to spend the time, effort, and money to enrich tools with domain knowledge if much of this knowledge is lost during every modification episode.

Another architectural issue impacting modification processes concerns support for design language versioning. In general this issue has been little considered with the exception of Stahl [111] and the Hermes perspective mechanism. However, as our design stories indicated, as design languages evolve, this evolution creates difficulties for past design products. In our design stories, designers spent much time retrofitting old design products to conform to new design languages. Often times, this entailed re-constructing the entire design from scratch using the new design language. Not only are designers not supported in this process, but oftentimes, old design products may no longer be readable by the design environment. This readability problem raises serious concerns for the stated DODE objectives of creating catalogs of re-usable designs and supporting long-term indirect communication with design rationale. Specifically, without support for design language versioning, older items in the catalog can probably not even be viewed in the evolved design environment. Thus, even though design rationale may still be readable, the designs or specific cases that inspired the rationale may not be. Stahl's perspective mechanism illustrated how versioning could be achieved in a shared object model architecture. However, research is required into applying

these ideas in the context of substrate-based architectures with all or partially distributed domain models.

Examining Alternative Object Models

To date, all DODEs but one have been based on object model formalisms using class inheritance mechanisms. One DODE – X-Network [106] – used an object formalism relying on prototype inheritance. Shipman claimed such an approach was necessary to support *in situ* incremental formalization processes. The operative phrase in this situation is *in situ*. As we saw in our design stories, a crucial part of supporting incremental formalization is allowing designers to refine concrete, individual instances embedded in design products instead of forcing them to operate immediately at the abstract class level.

Supporting instance level refinements appears to be an important requirement for supporting design language evolution. It particularly appears successful at promoting design-in-use, i.e., the intertwining of use and change of design languages, because it does not force excessive explicitness and formalization upon designers during the design (use) process. Object models supporting instance level refinements need to be investigated. It remains to be seen whether "pure" prototype inheritance models are required or hybrid approaches are sufficient. Shipman implemented his pure prototype inheritance model (HOS) from scratch. Others [45] have explored hybrid approaches by building models with some prototype inheritance properties using existing object systems based on class inheritance models.

Reconceptualizing the Construction Component

Chapters 5 and 6 discussed many issues surrounding the construction kit concept underlying the DODE construction component. In VDDE, several difficulties were encountered that highlight potentially problematic issues surrounding the current vision of construction kits. First, it proved difficult to strike a balance between expressiveness and facility. This was because abstractions representing a

middle ground between these two competing goals were difficult, if not impossible, to identify. Second, what should be represented in the construction kit and how it should be represented proved to be a fast moving target. It does not seem viable to provide designers in dynamic domains with predominantly canned representational systems. Designers need support for the *process* of creating the appropriate domain vocabularies and representations. Third, due to the rate of change and ownership issues, designers must be in control of this process of designing their design languages; they cannot rely on, nor necessarily want to depend on, system developer support in this particular area. Finally, construction kits as currently envisioned do not support *in situ* incremental formalization of design vocabularies. Construction kits introduce a formality barrier by requiring some degree of immediate formalization (i.e., at the time of input) when introducing new vocabulary items into the design language.

These issues indicate that conceptualizing the construction component in terms of providing designers with construction kits needs to be reconsidered. Specifically, experiences indicate that designers need to be provided with construction components more along the lines of meta-construction kits or kits for creating their own construction kit, rather than focusing on the end product. In this subsection, two recent research efforts along these lines – Visual AgenTalk and SmartMedia Tools – will be briefly considered with respect to how they address the issues outlined above.

Visual AgenTalk

Visual AgenTalk (VAT) is a new programming mechanism for Agentsheets [97] targeted at supporting a broader range of users to extend the functionality of Agentsheets' applications. VAT changes the roles of the system developer and the designer. Previously, the system developer created the top-level, domain-specific agents visible in the Agentsheets' gallery; the designer's role was simply to use these provided domain-specific agents. Using VAT, the system developer instead defines a set of more general agents and a set of VAT commands representing important programming primitives relevant to the

domain. Commands in VAT are small interactive language elements representing programming primitives that can be directly manipulated using drag/drop techniques. The designer uses the lower-level commands provided by the system developer to create new and modify existing domain-specific agents. In order to modify or extend the behavior of domain-specific agents, designers create rules by dropping commands into provided rule editors. In addition, designers can define when each rule gets tested and executed by specifying a trigger. A number of different triggers are provided to control the execution of complex simulation applications.

VAT could address several of the issues previously enumerated. Instead of system developers attempting to provide a complete set of language components at the "right" abstraction level, they instead provide a more general set of components and programming primitives that designers can use to create components that better suit changing design needs. Ideally, designers could perform many of these VAT extensions without developer support, thus reducing their overall dependence on system developers.

However, while VAT is a promising extension to the Agentsheets substrate, these possible improvements are very speculative in nature. For one thing, many end-user programming languages have been proposed that have met with limited success. Thus, it remains to be seen if designers are able and willing to use VAT as discussed. Second, there is still the challenge of enumerating a set of general components and VAT commands that are expressive enough to be able to solve typical design problems but facile enough to be usable by designers. This is the same problem encountered with the original construction kit approach but now pushed down to the next lower level. Finally, while VAT reduces the difficulty of defining new design units, it does not completely address the formality barrier previously discussed. New design units must still be formally defined at the time of input using VAT programming primitives and *in situ* instance-level elaborations are not supported.

SmartMedia Tools

A series of SmartMedia Tools [102, 114] combining positive aspects of both generic and domain specific applications have been developed. SmartMedia Tools enable practitioners to begin with generic graphic objects and to gradually enrich their tool with domain-specific vocabularies and relationships. Towards this end, SmartMedia Tools embody a specialized architecture and corresponding tools for refining domain objects both graphically and semantically (see Figure 7.2).

In generic graphics applications like MacDraw and flowcharting tools like Inspiration, graphics objects can be accessed and manipulated only at the visual interaction layer using direct manipulation. Similarly to these applications, SmartMedia Tools also provide a visual interaction layer supporting the direct manipulation of graphic objects. However, graphic objects in SmartMedia tools have two additional user-accessible representations: a system-supplied textual description of the object's presentation in the interface (the "presentation description") and a user-definable textual description of the object's meaning in the domain (the "semantic description"). The presentation description summarizes the visual properties of objects (e.g. shape, size,

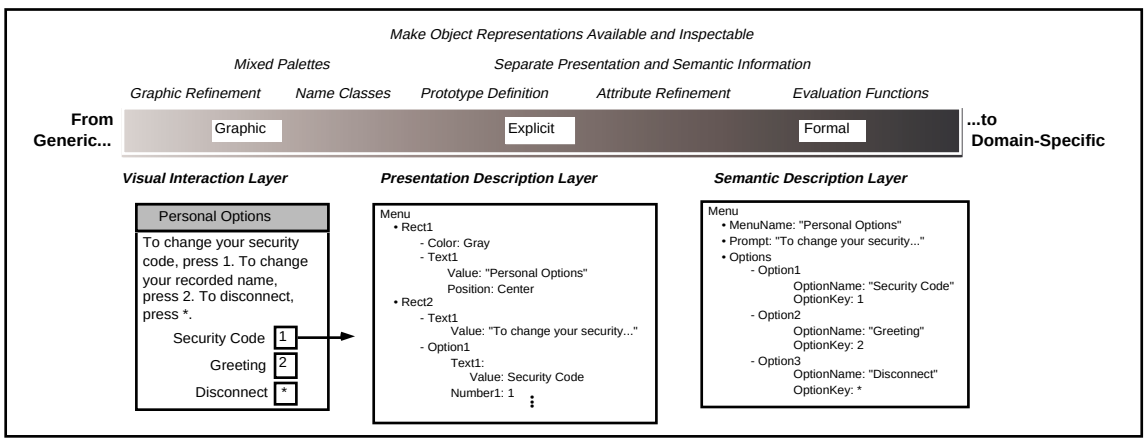

Figure 7.2: The SmartMedia architecture.

SmartMedia Tools consists of three layers: a visual interaction layer providing standard drawing tools, a presentation description layer, and a semantic description layer. In addition to supporting direct manipulation modifications, special editors are provided for modifying object descriptions at both the presentation and semantic levels.

color) and the semantic description specifies the domain-specific meaning of a graphic object (e.g. that a number represents a menu command). This description is saved for use by other applications in an open, object-oriented database.

Special editors are provided that enable users to inspect, extend, and modify the object descriptions directly. Relationships between parts of the presentation and semantic descriptions, such as evaluation functions, can be defined using spreadsheet-like functions. The representations are integrated in that changes in any of the object representations are reflected in all other object representations. Thus, changes performed by direct manipulation in the visual interaction layer are mirrored in the object's presentation description. At any time, users can select a graphic object in the visual interaction layer and define a class based on its properties. Once defined, objects are available in the palette for future reuse.

SmartMedia Tools could address several of the issues previously enumerated. SmartMedia Tools are designed to support designers to refine design languages all along the tacit to explicit to formal spectrum. Thus, SmartMedia Tools appear to lack the formality barrier found in more traditional construction kit approaches. Since, informal design languages are supported, the designers dependence on system developers is also reduced in that designers should be able to make many presentation-level and some shallow domain-semantic changes themselves. However, similar to VAT, it remains to be seen if designers are able and willing to use the end-user programming language provided to make deeper domain model extensions.

In summary, these two efforts emphasize moving beyond construction kits in different ways. Visual AgenTalk provides an end-user programming language extension to the Agentsheets' construction kit. The envisioned benefits include decreasing the designers' dependence on the system developer and increasing the flexibility of the construction kit. However, the functionality provided still focuses primarily on supporting designers to make explicit to

formal design language transitions; informal design languages are not addressed. SmartMedia Tools emphasize smoothly incorporating tacit to explicit to formal transitions by enabling design language refinements at both the presentation and semantic levels. However, the end-user programming language provided for making deeper domain model extensions is a dialect of Lisp and may be too low-level and difficult for designers to use. Thus, further research beyond these two promising projects is still required to address the issues surrounding construction kits in a more comprehensive manner.

BRIDGING THE GAP

As previously discussed, the goal is to make systems that are flexible enough to accommodate the continual evolution of design languages, yet capable of providing support for using the design language in its current state. Two types of evolutionary change were identified and characterized in this dissertation: continual incremental *elaborations* and occasional radical *modifications* to existing design language. There are several difficult challenges to overcome to support these two types of evolution.

Firstly, to support design language elaboration, the key challenge is providing incremental approaches that enable designers to perform most of the elaborations themselves. Furthermore, these approaches must allow designers to begin with informal and ill-defined design languages and to gradually refine them through the entire spectrum of design language explicitness; i.e. to progressively transition from tacit to explicit to formal design languages. As we saw in this dissertation, generic tools favor the tacit to explicit end of the spectrum by supporting mainly presentation-level refinements while domain-oriented tools favor the formal end of the spectrum by supporting mainly semantic-level refinements.

Secondly, with respect to modification processes, the key difficulty is to support radical change without invalidating large amounts of existing domain models or existing design products. If domain

vocabulary definitions and established relationships are lost during every modification event, designers may decide it is not worth the effort to invest systems with this domain knowledge in the first place. As we saw in our three design stories, this is especially important in dynamic and innovative domains where radical modification events occur with frequency. To support modification processes without domain knowledge loss requires flexible software architectures, new forms of integration services, and more flexible object-oriented formalisms. As yet, neither Toolbelt or DODE models address this particular challenge.

In the introduction of this chapter it was proposed that designers need tools that combine elements of both generic and domain-oriented systems. To expand on this statement, design environments ideally would turn the current dichotomy of generic versus domain-oriented systems into a seamless continuum and provide:

(1) the rich, flexible formatting and editing features associated with generic tools,

(2) the supportive functionality offered by domain-specific tools,

(3) design language elaboration mechanisms that support incrementally bridging the gap between the two endpoints, and

(4) flexible underlying architectures and infrastructures that can accommodate larger-scale evolutionary changes.

Suggestions were made concerning how toolbelts could start to bridge this gap by providing more support for design language elaboration mechanisms. However, the potential to change this model is limited, particularly at the architectural level. Thus, it is doubtful if this ideal will be realized by the Toolbelt model.

Suggestions were also made concerning how DODEs could start to bridge this gap. The suggestions considered changing construction

components to incorporate some of the functionality associated with generic tools and investigating new architectures and infrastructure to support design language modification processes. Since the DODE model is an active research area, the potential to move it towards the ideal is much higher.

However, it remains to be seen if these existing models are flexible enough to accommodate the ideals outlined above; in the end, new computational models not envisioned here may be required.

SUMMARY

This chapter discussed possible areas for future work in the context of both Toolbelt and DODE models. In the Toolbelt model, two areas for re-designing tools were considered: improving support for articulating domain vocabularies and improving support for establishing relationships. Several suggestions for possible forms of organizational support were suggested to address problems observed in locating and extending tools. In the DODE model, three areas requiring further research were discussed. New architectures need to be considered that better support design language modification processes. Alternative object-oriented formalisms need to be examined that could better support incremental design language elaboration processes. And, the use of construction kits as the basis for the DODE construction component needs to be reconsidered before applying the DODE model to future dynamic and innovative design domains.

CHAPTER 8

CONCLUSIONS

This dissertation considered the problem of providing small, independent design teams working in innovative and dynamic domains with computational design support tools. The emphasis of this research was on understanding the evolutionary patterns of change inherent in dynamic domains and investigating how design tools could support these patterns. The motivating problem investigated in this thesis was design environment flexibility, specifically:

> How can design environments provide domain-oriented support for daily design activities *yet still be flexible enough* to accommodate evolutionary changes in the domain that occur over time?

This chapter begins by summarizing the approach used to investigate this question. Next, the findings of this research are reviewed. Finally, this dissertation concludes by enumerating the contributions of this research.

SUMMARY OF APPROACH

The approach used in this research was divided between empirical studies, theory development, system building, and system assessment (Figure 8.1). Empirical studies of three design groups were conducted to better understand the patterns of change in the communities' design products, tools, and practices occurring over extended periods of time. Analyses of these studies were used to characterize in detail the observed evolutionary processes and provided the motivation for the

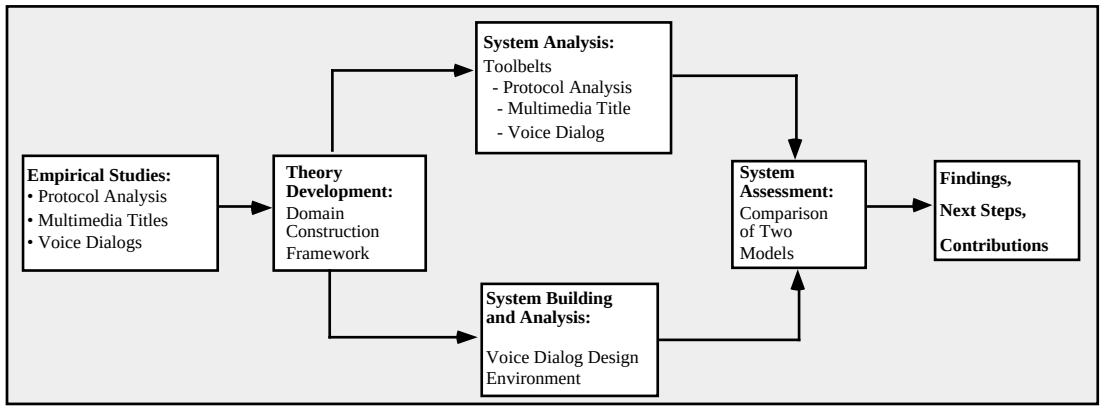

Figure 8.1. Overview of Research Approach

theoretical contribution: the domain construction framework. This framework was then used to analyze and compare two computational models at seemingly different ends on the support / flexibility spectrum. Our initial intuitions were that the Toolbelt model, where practitioners assemble and evolve collections of generic software applications, offered more flexibility. The DODE model, where customized domain-oriented tools are created for a specific community, seemingly provides better support but is less flexible.

Specific design environments based on these two different computational models were developed and analyzed during the course of this research. Three systems, based on the Toolbelt model, were created over time by the designers themselves and studied as part of this research. The Voice Dialog Design Environment (VDDE), based on the domain-oriented design environment model, was collaboratively constructed by myself and one group of designers (who were also concurrently involved in Toolbelt construction). VDDE enables designers to construct and simulate flow-chart like representations of phone-based interfaces. An embedded critiquing component analyzes these representations for compliance with various user interface guidelines and consistency with related design products.

This comparison looked at which particular aspects of these models helped or hindered their use and overall flexibility. The purpose of this analysis was to investigate our initial intuitions in order to better understand which particular aspects of a model contributed to its

flexibility. The findings of this analysis were used to guide recommendations for the next generation of design support environments.

SUMMARY OF FINDINGS

The findings of this dissertation fall into two categories: empirical findings concerned with characterizing macro design processes and system-oriented findings related to the analyses of computational models and design environments.

Empirical Findings

Studies of three design communities were used to understand long-term changes in design tools, practices and products. These groups all worked in different areas of user interface and software design: protocol analysis, multimedia title design, and voice dialog design. The workplace settings were very different, ranging from a university team doing a short, one-off design project to industrial designers working on a continual stream of upgrades to an existing product line.

Our empirical analyses indicate that the central constituent of a domain is the shared understanding between domain practitioners that enables a community to use tools in a practiced way to create design products in a consistent manner. This shared understanding takes a tangible form over time called a "design language." While creating individual design products (use of a design language), designers change the design language used to create these products (evolution). Specifically, design communities gradually construct and evolve their design language over time by: (1) defining important domain objects, (2) creating and evolving multiple representations for viewing these objects, and (3) establishing relationships between objects and representations. This observed design-in-use process was labeled *domain construction*.

Three basic macro-scale patterns of evolutionary change to design languages were identified and characterized: maintenance, elaboration,

and modification. Maintenance occurs when individuals enact design languages using the (sometimes implicit) rules of substance and form without alteration. Elaboration occurs when individuals consistently but slightly adapt design languages to reflect new conditions. Modification of design languages occurs when individuals depart significantly and persistently from existing rules, such as when prose documents are replaced by tabular representations. The subprocesses involved in these larger-scale patterns of change were characterized in detail to create the domain construction analytical framework.

System-Oriented Findings

Two existing computational models – Toolbelts and DODEs – were analyzed according to the domain construction framework. These analyses revealed that our initial intuitions about both models were only partially correct; both Toolbelts and DODEs provided mixed support for different aspects of use and evolutionary change. Toolbelts fell short at supporting some aspects of use, such as design evaluation, due to their overly generic nature. On the other hand, DODEs fell short at supporting some aspects of use, such as design construction, because of their overly narrow domain-orientation. Both models provided mixed support for evolutionary processes due to limitations and inflexibilities in their underlying data models, integration services, and architecture.

Thus, the analyses indicated that the question of whether generic or domain-specific software is better suited for supporting use and evolutionary change is probably too simplistic. In fact, what these designers need are tools that bridge the gap between these two extremes by combining elements of both. To bridge this gap, design environments need a specialized architecture, a flexible object model, and corresponding tools for refining domain objects both graphically and semantically. Specifically, tools should support the domain construction process by enabling practitioners to begin with generic objects and to gradually enrich their design environment with domain-specific vocabularies and relationships.

Finally, our analyses show that the question of who is in control of the domain construction process is a critical issue, both from a pragmatic rate of change perspective and a political empowerment perspective. When trying to support politically-empowered designers in dynamic domains, it is important that environments enable designers to be in control of the design language evolution process.

CONTRIBUTIONS OF THIS RESEARCH

The contributions of this research were three-fold. One goal was to contribute to our *empirical* understanding of macro design processes; i.e., the evolutionary patterns of change in artifacts and representations used by design communities over extended periods of time. This goal was met by studying the three design communities and characterizing the domain construction process in terms of design language use, elaboration, and modification.

The second goal was to contribute to *design theory* by extending existing theories to account for the observed macro design processes. This goal was met by analyzing in detail the subprocesses involved in use, elaboration, and modification and relating these subprocesses to existing theories of design in order to create the domain construction analytical framework.

The third goal was to contribute to computational models of *design environments* by deriving principled criteria for analyzing and selecting between different architectural approaches, object models, and tool-development substrates when creating design environments for a particular domain. This goal was partially met by analyzing the Toolbelt and DODE computational models from the perspective of the domain construction framework. These analyses did not provide specific guidelines for making a selection between architectural features but, instead, contributed to the goal by indicating what architectural and system design issues need to be considered. When embarking on creating a design environment, issues such as flexibility of the object model and the software architecture promoted by possible tool or

component choices should be carefully considered. As shown in this dissertation, choices made at lower, infrastructure levels of the design environment deeply affect the functionality and flexibility available to designers.

Overall, these goals contribute towards answering the motivating problem by indicating promising directions for the next generation of design environments in order to better meet the paradoxical goals of providing support and flexibility at the same time.

REFERENCES

1. Apple Computer, Inc., *Macintosh Common Lisp Reference,* 1992.

2. Arango, G. and R. Prieto-Diaz, "Introduction," in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Ed., IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 9-32.

3. Bell, B., "Using Programming Walkthroughs to Design a Visual Language," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 1992.

4. Bell, B., W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde and B. Zorn, "The Programming Walkthrough: A Structured Method for Assessing the Writability of Programming Languages," *Technical Report,* CU-CS-577-92, Dept. of Computer Science, University of Colorado at Boulder, 1992.

5. Blomberg, J. L. and A. Henderson, "Reflections on Participatory Design: Lessons from the Trillium Experience," *Human Factors in Computing Systems (CHI '90)*, 1990, pp. 353-359.

6. Bobrow, D. G., L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales and D. A. Moon, "Common Lisp Object System Specification," *Document,* 88-002R, 1988.

7. Bodker, S. and K. Gronbaek, "Design in Action: From Prototyping by Demonstration to Cooperative Prototyping," in *Design at Work: Cooperative Design of Computer Systems*, J. Greenbaum and M. Kyng, Ed., Lawrence Erlbaum Associates, Hillsdale, NJ, 1991, pp. 197-218.

8. Bonnardel, N., "Criteria Used for Evaluation of Design Solutions," *Designing for Everyone and Everybody (Proceedings of the 11th Congress of the International Ergonomics Association)*, Paris (July), 1991, pp. 1043-1045.

9. Bonnardel, N. and T. Sumner, "From System Development to System Assessment: Exploratory Study of the Activity of Professional Designers," *7th European Conference on Cognitive Ergonomics - ECCE'7, Human-Computer Interaction: From individuals to groups in work, leisure, and everyday life.*, Bonn, Germany (Sept. 5-8), 1994, pp. 23-36.

10. Bonnardel, N. and T. Sumner, "Supporting Evaluation in Design: The Impact of Critiquing Systems on Designers of Different Skill Levels," *To appear in: ACTA Psychologica,* Special Issue on Cognitive Ergonomics, 1996.

11. Brancheau, J. C. and C. V. Brown, "The Management of End-User Computing: Status and Directions," *ACM Computing Surveys,* Vol. 25, pp. 437-482, 1993.

12. Brown, J. S. and P. Duguid, "Borderline Issues: Social and Material Aspects of Design," *Human-Computer Interaction,* Vol. 9, pp. 3-36, 1994.

13. Carroll, J. M. and M. B. Rosson, "Paradox of the Active User," in *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, MIT Press, Cambridge, MA, 1987, pp. 80-111.

14. Conklin, J. and M. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," *Transactions of Office Information Systems,* Vol. 6, pp. 303-331, 1988.

15. Curtis, B., H. Krasner and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM,* Vol. 31, pp. 1268-1287, 1988.

16. Davis, R., "Expert Systems: How Far Can They Go," *AI Magazine,* Vol. 10, pp. 61-67, 1989.

17. DiGiano, C. and M. Eisenberg, "Self-Disclosing Design Tools: A gentle introduction to end-user programming," *Symposium on Designing Interactive Systems (DIS '95)*, Ann Arbor, MI (August 23-26), 1995, pp. 189-197.

18. DiGiano, C., C. Lewis and C. Hurtt, "The Future of Programming Interactive Experience," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 49-50.

19. Dunlop, C. and R. Kling, "Introduction to Part III: Computerization and the Tranformation of Work," in *Computerization and Controversy*, Academic Press, New York, 1991, pp. 182-199.

20. Ehn, P., *Work-Oriented Design of Computer Artifacts,* arbetslivscentrum, Stockholm, 1989.

21. Eisenberg, M., "Programmable Applications: Interpreter meets Interface," *A.I. Memo,* 1325, MIT Artificial Intelligence Laboratory, 1991.

22. Eisenberg, M. and G. Fischer, "Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 431-437.

23. Eisenberg, M. and A. Nishioka, "Creating Polyhedral Models by Computer," *To appear in: Journal of Computers in Mathematics and Science Teaching,*, 1996.

24. Feiler, P. H. and K. C. Wallnau, "Tool Integration and Environment Architectures," *Technical Report*, CMU/SEI-91-TR-11, Software Engineering Institute, 1991.

25. Fischer, G., "Domain-Oriented Design Environments," in *Automated Software Engineering*, Kluwer Academic Publishers, Boston, MA., 1994, pp. 177-203.

26. Fischer, G. and A. Girgensohn, "End-User Modifiability in Design Environments," *Human Factors in Computing Systems (CHI'90)*, Seattle, WA (April 1-5), 1990, pp. 183-191.

27. Fischer, G., A. Lemke, T. Mastaglio and A. Morch, "Using Critics to Empower Users," *Human Factors in Computing Systems (CHI'90)*, Seattle, WA (April 1-5), 1990, pp. 337-347.

28. Fischer, G. and A. C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *Human-Computer Interaction,* Vol. 3, pp. 179-222, 1988.

29. Fischer, G., A. C. Lemke, T. Mastaglio and A. Morch, "The Role of Critiquing in Cooperative Problem Solving," *ACM Transactions on Information Systems,* Vol. 9, pp. 123-151, 1991.

30. Fischer, G., A. C. Lemke, R. McCall and A. Morch, "Making Argumentation Serve Design," *Human Computer Interaction,* Vol. 6, pp. 393-419, 1991.

31. Fischer, G., R. McCall and A. Morch, "Design Environments for Constructive and Argumentative Design," *Human Factors in Computing Systems (CHI '89)*, Austin, Texas (May), 1989, pp. 269-275.

32. Fischer, G., R. McCall, J. Ostwald, B. Reeves and F. Shipman, "Seeding, Evolutionary Growth and Reseeding: Supporting the Incremental Development of Design Environments," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 292-298.

33. Fischer, G., K. Nakakoji, J. Ostwald, G. Stahl and T. Sumner, "Embedding Computer-Based Critics in the Contexts of Design," *Human Factors in Computing (Interact '93 and CHI '93)*, Amsterdam (24-29 April), 1993, pp. 157-164.

34. Fischer, G., K. Nakakoji, J. Ostwald, G. Stahl and T. Sumner, "Embedding Critics in Design Environments," *The Knowledge Engineering Review,* Vol. 8, pp. 285-307, 1993.

35. Galegher, J. and R. Kraut, "Computer-Mediated Communication and Collaborative Writing: Media Influence and Adaption to Communication Constraints," *CSCW '92: Conference on Computer-Supported Cooperative Work*, Toronto, Canada, 1992, pp. 155-162.

36. Gantt, M. and B. Nardi, "Gardeners and Gurus: Patterns of Cooperation Among CAD Users," *Human Factors in Computing Systems (CHI '92)*, Monterey, CA (May 3-7), 1992, pp. 107-117.

37. Garlan, D. and M. Shaw, "An Introduction to Software Architecture," *Technical Report,* CMU/SEI-94-TR-21, Software Engineering Institute, 1994.

38. Girgensohn, A., "End-User Modifiability in Knowledge-Based Design Environments," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science (Technical Report: CU-CS-595-92), 1992.

39. Goodman, D., *Danny Goodman's HYPERCARD Developer's Guide,* Bantam Books, Toronto, 1988.

40. Gould, J. D., S. J. Boies and C. Lewis, "Making Usable, Useful, Productivity - Enhancing Computer Applications," *Communications of the ACM,* Vol. 34, pp.74-85, 1991.

41. Green, T. R. G., "Cognitive Dimensions of Notations," *People and Computers V*, Nottingham, United Kingdom, 1989, pp. 443-460.

42. Green, T. R. G., "The Cognitive Dimension of Viscosity: a sticky problem for HCI," *Human-Computer Interaction – INTERACT '90*, 1990, pp. 79-86.

43. Greenbaum, J. and M. Kyung, *Design at Work: Cooperative Design of Computer Systems,* Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.

44. Gronbaek, K., J. Grudin, S. Bodker and L. Bannon, "Achieving Cooperative System Design: Shifting From a Product to a Process Focus," in *Participatory Design: Principles and Practices*, D. Schuler and

A. Namioka, Ed., Lawrence Erlbaum Associates, Hillsdale, NJ, 1993, pp. 79-97.

45. Gross, M., Using MCL to Create a Prototype Inheritance Object Model, Personal Communication, 1996

46. Grudin, J., "Seven Plus One Challenges in Understanding Social Dynamics for Groupware Developers," *Human Factors in Computing Systems (CHI '91)*, New Orleans, LA (April 27 - May 2), 1991, Tutorial.

47. Grudin, J., "Evaluating Opportunities for Design Capture," *Technical Report,* , University of California, Irvine, 1992.

48. Guindon, R., "Designing the Design Process: Exploiting Opportunistic Thoughts," *Human Computer Interaction,* Vol. 5, pp. 305-344, 1990.

49. Guindon, R., "Knowledge Exploited by Experts during Software System Design," *International Journal of Man-Machine Studies, Special Issue: What Programmers Know,* Vol. 33, pp. 279-304, 1990.

50. Guindon, R., "Requirements and Design of DesignVision, An Object-Oriented Graphical Interface to an Intelligent Software Design Assistant," *Human Factors in Computing Systems (CHI '92)*, Monterey, CA (May 3-7), 1992, pp. 499-506.

51. Guindon, R., H. Krasner and B. Curtis, "Breakdowns and Processes During the Early Phases of Software Design by Professionals," in *Empirical Studies of Programmers: Second Workshop*, G. Olson, S. Sheppard and E. Soloway, Ed., Ablex Publishing Corporation, Norwood, New Jersey, 1987, pp. 65-82.

52. Halasz, F., "Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems," *Communications of the ACM,* Vol. 31, pp. 836-852, 1988.

53. Halstead-Nussloch, R., "The Design of Phone-Based Interfaces for Consumers," *Human Factors in Computing Systems (CHI'89)*, Austin, Texas, 1989, pp. 347-352.

54. Hansen, W., "Introduction to User Interface Systems for HCI Developers and Researchers," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 377-378.

55. Harstad, B., "New Approaches to Critiquing: Pluralistic Critiquing, Consistency Critiquing, and Multiple Intervention Strategies," University of Colorado at Boulder, Masters dissertation, Dept. of Computer Science, 1993.

56. Henderson, A. and M. Kyng, "There's No Place Like Home: Continuing Design in Use," in *Design at Work: Cooperative Design of Computer Systems*, M. Kyng and J. Greenbaum, Ed., Lawrence Erlbaum Associates, Hillsdale, NJ, 1991, pp. 219-240.

57. Hill, W. C., "The Mind at AI: Horseless Carriage to Clock," *AI Magazine,* Vol. 10, pp. 29-41, 1989.

58. Johnson, J., "Selectors: Going Beyond User-Interface Widgets," *Human Factors in Computing Systems (CHI '92)*, Monterey, CA (May 3-7), 1992, pp. 273-279.

59. Johnson, J., B. A. Nardi, C. L. Zarmer and J. Miller, "ACE: Building Interactive Graphical Applications," *Communications of the ACM,* Vol. 36, pp. 40-55, 1993.

60. Keil-Slawik, R., "Artifacts in Software Design," in *Software Development and Reality Construction*, C. Floyd, H. Zullighoven, R. Budde and R. Keil-Slawik, Ed., Springer-Verlag, Berlin, 1992, pp. 168-188.

61. Kolodner, J., "Improving Human Decision Making through Case-Based Decision Aiding," Vol. 12, pp. 52-68, 1991.

62. Kolodner, J., *Case-Based Reasoning,* Morgan Kayfmann, San Mateo, CA, 1993.

63. Landauer, T. K., *The Trouble with Computers: Usefulness, Usability, and Productivity,* MIT Press, Cambridge, MA, 1995.

64. Lewis, C. and G. M. Olson, "Can Principles of Cognition Lower the Barriers to Programming?," *Empirical Studies of Programmers: Second Workshop*, G. Olson, S. Sheppard and E. Soloway, Ed., Ablex Publishing Corporation, Norwood, New Jersey, 1987,pp. 248-263.

65. Lewis, C., J. Rieman and B. Bell, "Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System," *Human-Computer Interaction,* Vol. 6, pp. 319-355, 1991.

66. Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *First Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '86)*, Portland, OR (Sept. 29 - Oct. 2), 1986, pp. 214-223.

67. Mackay, W., "Patterns of Sharing of Customizable Software," *CSCW'90: Conference on Computer-Supported Cooperative Work*, Los Angeles, CA, 1990, pp. 209-221.

68. Mackay, W., "Users and Customizable Software: A Co-Adaptive Phenomena," Massachusetts Institute of Technology: Cambridge, MA, Ph.D. dissertation, 1990.

69. Mackay, W., "Triggers and Barriers to Customizing Software," *Human Factors in Computing Systems (CHI '91)*, New Orleans, LA (April 27 - May 2), 1991, pp. 153-160.

70. MacLean, A., K. Carter, L. Lovstrand and T. Moran, "User-tailorable Systems: Pressing the issues with Buttons," *Human Factors in Computing Systems (CHI '90)*, Seattle, WA (April 1-5), 1990, pp. 175-182.

71. Madsen, K. H. and P. H. Aiken, "Experiences Using Cooperative Interactive Storyboard Prototyping," *Communications of the ACM*, Vol. 36, pp. 57-64, 1993.

72. McCall, R., "PHIBIS: Procedurally Hierarchical Issue-Based Information Systems," *Conference on Architecture at the International Congress on Planning and Design Theory*, 1987.

73. McLean, E. R., L. A. Kappelman and J. P. Thompson, "Converging End-User and Corporate Computing," *Communications of the ACM*, Vol. 36, pp. 79-92, 1993.

74. Microsoft Corp., *Object Linking and Embedding (OLE),* Part No. 098-31727, Redmond, WA., 1992.

75. Morch, A., "Three Levels of End-User Tailoring: Customization, Integration, and Extension," *Computers in Context: Joining Forces in Design (Third Decennial Conference)*, Aarhus, Denmark (August 14-18), 1995, pp. 157-166.

76. Nakakoji, K., "Increasing Shared Understanding of a Design Task Between Designers and Design Environments: The Role of a Specification Component," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science (Technical Report: CU-CS-651-93), 1993.

77. Nakakoji, K. and G. Fischer, "Catalog Explorer: Exploiting the Synergy of Integrated Design Environments," *Software Symposium'90 (Kyoto, Japan),* , pp. 264-271, 1990.

78. Nakakoji, K., T. Sumner and B. Harstad, "Perspective-Based Critiquing: Helping Designers Cope with Conflicts among Design Intentions," *Artificial Intelligence in Design '94*, Lausanne, Switzerland (August), 1994, pp. 449-466.

79. Nardi, B. A., "Beyond Models and Metaphors: Visual Formalisms in User Interface Design," *Journal of Visual Languages and Computing,* Vol. 4, pp. 5-33, 1993.

80. Nardi, B. A., *A Small Matter of Programming,* The MIT Press, Cambridge, MA, 1993.

81. Nardi, B. A. and J. A. Johnson, "User Preferences for Task-specific vs. Generic Application Software," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 392-398.

82. Norman, D. A., *Things That Make Us Smart,* Addison-Wesley Publishing Company, Reading, MA, 1993.

83. Orlikowski, W. J. and J. J. Baroudi, "Studying Information Technology in Organizations: Research Approaches and Assumptions," *Information Systems Research,* Vol. 2, pp. 1-29, 1991.

84. Ostwald, J., "The Evolving Artifact Approach to System Development," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 1996.

85. Peters, T., "Re-Inventing Civilisation," *CROSSBORDER, the Economist Intelligence Unit's Journal of Multinational Management,* Vol. Summer, pp. 12-15, 1993.

86. Petre, M., "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming," *Communications of the ACM,* Vol. 38, pp. 33-44, 1995.

87. Piersol, K., "Under the Hood: A Close-Up of OpenDoc," *BYTE,* Vol. 19, pp. 183-188, 1994.

88. Polanyi, M., *The Tacit Dimension,* Doubleday, Garden City, NY, 1966.

89. Prieto-Diaz, R. and P. Freeman, "Classifying Software for Reusability," Vol. 4, pp. 6-16, 1987.

90. Quinn, J. B., *Intelligent Enterprise,* The Free Press, New York, N.Y., 1992.

91. Reeves, B. N., "Supporting Collaborative Design by Embedding Communication and History in Design Artifacts," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science (Technical Report: CU-CS-651-93), 1993.

92. Reitman, W. R., *Cognition and Thought: An Information Processing Approach.,* Wiley, New York, 1965.

93. Repenning, A., "The OPUS User Manual," *Technical Report*, CU-CS-556-91, Dept. of Computer Science, University of Colorado at Boulder, 1991.

94. Repenning, A., "Agentsheets: A Tool for Building Domain-Oriented, Dynamic, Visual Environments," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science (Technical Report: CU-CS-693-93), 1993.

95. Repenning, A., "Programming Substrates to Create Interactive Learning Environments," *Journal of Interactive Learning Environments,* Vol. 4 (Special Issue on End-User Environments), pp. 45-74, 1994.

96. Repenning, A., "Bending the Rules: Steps toward Semantically enriched Graphical Rewrite Rules," *Proceeding of Visual Languages*, Darmstadt, Germany, 1995.

97. Repenning, A. and J. Ambach, "Tactile Programming: Unifying the Application World and the Programming World," *Submitted to: Advanced Visual Interfaces (AVI '96)*, Gubbio, Italy, 1996.

98. Repenning, A. and T. Sumner, "Using Agentsheets to Create a Voice Dialog Design Environment," *Symposium on Applied Computing (SAC '92)*, Kansas City, MO., 1992, pp. 1199-1207.

99. Repenning, A. and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer (Special Issue on Visual Programming),* Vol. 28, pp. 17-25, 1995.

100. Rieman, J. F., "Learning Strategies and Exploratory Behavior of Interactive Computer Users," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 1994.

101. Rittel, H. and M. M. Webber, "Planning Problems are Wicked Problems," in *Developments in Design Methodology*, N. Cross, Ed., John Wiley & Sons, New York, 1984, pp. 135-144.

102. Schneider, K. and M. Stolze, "SMArT CASE: Supporting Co-Improvement of Process, Tools, and Notations," *Submitted to: International Conference on Software Engineering (ICSE-18)*, Berlin, Germany (March 26-30), 1996.

103. Schoen, D. A., *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.

104. Schuler, D. and A. Namioka, *Participatory Design: Principles and Practices,* Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.

105. Shaw, M., "Software Architecture for Shared Information Systems," *Technical Report,* CMU/SEI-93-TR-3, Software Engineering Institute, 1993.

106. Shipman, F. M., "Supporting Knowledge-Base Evolution with Incremental Formalization," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 1993.

107. Shipman, F. M. and R. McCall, "Supporting Knowledge-Base Evolution with Incremental Formalization," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 285-291.

108. Silverman, B., "Survey of Expert Critiquing Systems: Practical and Theoretical Frontiers," *Communications of the ACM,* Vol. 35, pp. 106-127, 1992.

109. Simon, H. A., *The Sciences of the Artificial,* The MIT Press, Cambridge, MA, 1981.

110. Solomon, C., "Integrating Office Applications with OLE," in *Developing Applications with Microsoft Office*, Ed., Microsoft Press, Redmond, Washington, 1995, pp. 387-450.

111. Stahl, G., "Interpretation in Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 1993.

112. Steele, G. L., *Common LISP: The Language (2nd Edition),* Digital Press, Burlington, MA, 1990.

113. Stefik, M. J. and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine,* Vol. 6, pp. 1986.

114. Stolze, M. and T. Sumner, "SmartMedia Tools: Bridging the Gap Between Generic Applications and Domain-Oriented Systems," *Technical Report*, CU-CS-792-95, Dept. of Computer Science, University of Colorado at Boulder, 1995.

115. Sullivan, J., "A Proactive Computational Approach to Learning While Working," University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 1994.

116. Sumner, T., "The High-Tech Toolbelt: A Study of Designers in the Workplace," *Human Factors in Computing Systems (CHI '95)*, Denver, CO (May 7-11), 1995, pp. 178-185.

117. Sumner, T., S. Davies, A. C. Lemke and P. G. Polson, "Iterative Design of a Voice Dialog Design Environment," *Technical Report,* CU-CS-546-91, Dept. of Computer Science, University of Colorado at Boulder, 1991.

118. Sumner, T., C. Marra and C. Lewis, "Results from Programming Walkthroughs are Wildly Inconsistent, But...," *Technical Report*, CU-CS-745-94, Dept. of Computer Science, University of Colorado at Boulder, 1993.

119. Sumner, T. and M. Stolze, "Evolution, Not Revolution: PD in the Toolbelt Era," *Computers in Context: Joining Forces in Design (CIC '95)*, Aarhus, Denmark (August 14-18), 1995, pp. 30-39.

120. Trigg, R. and S. Bodker, "From Implementation to Design: Tailoring and the Emergence of Systemization in CSCW," *Conference on Computer Supported Cooperative Work (CSCW '94)*, Chapel Hill, North Caroline (October 22-26), 1994, pp. 45-54.

121. Udell, J., "Beyond DOS: Visual Basic Custom Controls Meet OLE," *BYTE,* Vol. 19, pp. 197-200, 1994.

122. Udell, J., "Componentware," *BYTE,* Vol. 19, pp. 46-56, 1994.

123. Winkler, D. and S. Kamins, *Hypertalk 2.0: The Book,* Bantam Books, Toronto, 1990.

124. Winograd, T. and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design,* Addison-Wesley, Menlo Park, CA, 1986.

125. Winston, P. H., *Artificial Intelligence,* Addison-Wesley, Reading, MA, 1984.

126. Yates, J. and W. Orlikowski, "Genres of Organizational Communication: A Structurational Approach to Studying Communication and Media," *Academy of Management Review,* Vol. 17, pp. 299-326, 1992.

127. Yin, R., *Case Study Research: Design and Methods,* SAGE Publications, Newbury Park, CA, 1984.

128. Zarmer, C. L. and C. Chew, "Frameworks for Interactive, Extensible, Information-Intensive Applications," *Symposium on User Interface Software and Technology (UIST '92)*, Monterey, CA (November 15-18), 1992, pp. 33-41.